Study Notes for AI (CS 440/520)
Lecture 5: Local Search

Corresponding Book Chapters: 4.1-4.2-4.5
Note: These notes provide only a short summary and some highlights of the material covered in the corresponding lecture based on notes collected from students. Make sure you check the corresponding chapters. Please report any mistakes in or any other issues with the notes to the instructor.

1 Local Search Algorithms

The search algorithms discussed so far have exponential time and/or space complexities because they store information describing the entire path that is being explored and explore all paths exhaustively. In some problems (i.e., going from Arad to Bucharest using the road network of Romania), this path is necessary to describe the solution. In many cases, however, the path is irrelevant and only the final solution is desired. In the 8-queen Problem for example, the solution involves only the final positions of the queens. When the path is not important, local search algorithms can greatly improve time and space complexity by making decisions based only local information. Instead of remembering every location traveled, the algorithm must only remember the current location.

- Local search characteristics:
  - Used when the path is not important
  - Remembers only a single/few current states
  - Moves only between neighboring nodes
  - Can be applied to maximization problems
- Advantages:
  - Good space complexity (constant)
  - Can find reasonable solutions, even in very large or infinite state spaces
  - Can solve problems where there is no explicit notion of a goal
  - Easy to code
- The drawback of local search techniques is that they can get "stuck" in:
  - Local maxima: The algorithm only finds the first solution that is locally available. This may not be the best overall solution.
  - Plateaux: If a neighboring value is equal to the current value, the algorithm will not explore that direction, even if better values lie just beyond it.

1.1 Hill Climbing

Hill Climbing is a greedy local search algorithm that selects the best choice among all neighboring states. The operation of hill climbing is described in Figure 1. The states in a search space can be represented as a position in one or more dimensions, where each position has an associated fitness (fitness is represented in Figure 1 using the vertical axis). Neighboring states are those that can be accessed directly from the current state. Hill climbing is accomplished by iteratively choosing neighboring states that have the highest available fitness. This allows the fitness of the current state to progressively improve until it is the best of all nearby states.

Algorithm 1 `Hill Climbing`

```
current_node ← MAKE_NODE(initial_state)
loop
  neighbor ← the best successor of current_node
  if value(neighbor) ≤ value(current_node) then
    current_node ← neighbor
  else
    return current_node
```

- Example: Queens problem (see Figure 2)
  - The evaluation function is defined as the number of queen-pairs, which conflict with each other
  - Numbers on the grid represent the resulting fitness if a queen in the corresponding column is moved to that location
  - The evaluation function is optimized by moving a queen to the square containing the smallest value
  - The evaluation function is recalculated for every square after each move
  - Queens are moved in this manner until the evaluation function can no longer be improved
- Stats: 8-Queen problem using hill climbing
  - There are 17,000,000 possible states
  - Problem is solved → 14% of the time after an average of 4-steps
  - Local minimum is found → 86% of the time after an average of 4-steps

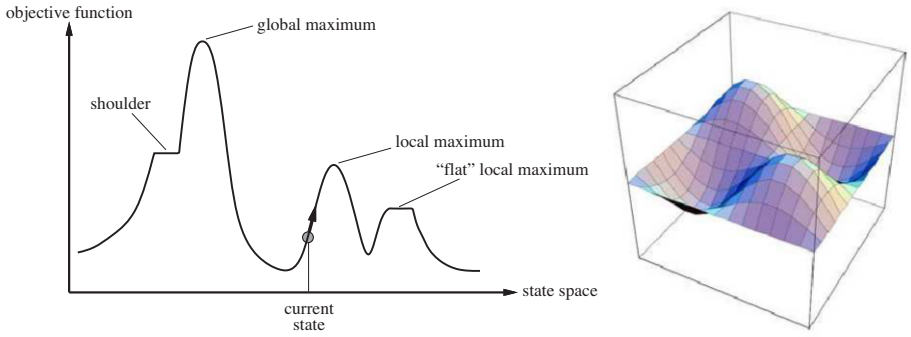

Figure 1: Hill climbing attempts to "climb the gradient" of an evaluation function.

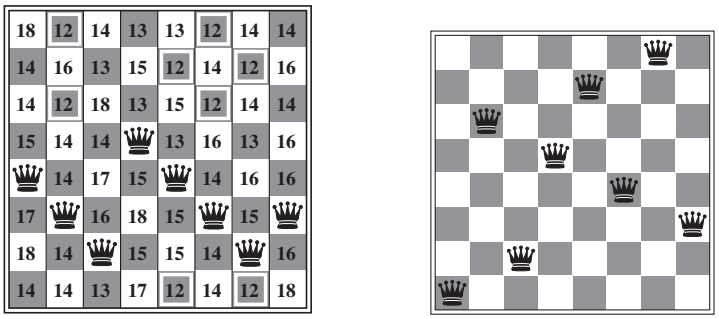

Figure 2: (left) Evaluation function for the 8-queen problem that allows the application of hill-climbing. (right) A local minimum state.

1.1.1 Avoiding Plateaux

Plateaux problems can be overcome by introducing "random walks":

- If neighboring locations have equal fitness, randomly select one of them and continue
  - Problem: this can cause infinite looping
  - Solution: Thresholding the number of allowed moves. If beyond threshold, break out of the loop
- Stats: 8-queen problem with random-walk:
  - Problem is solved $\rightarrow$ 94% of the time after an average of 21 steps
  - Problem fails $\rightarrow$ 6% of the time after an average of 64 steps
- Complexity:
  - Time complexity: O(d) where "d" is the longest path to the solution
    - Note: ∞ is possible
  - Space complexity: O(b) where "b" is the branching factor
    - If "b" is considered constant, then O(1)

1.1.2 Avoiding Local Extrema

Local extrema can be overcome using "restarts"

- If the algorithm gets stuck, reinitialize to a random configuration, and try again
- Stats: 8-queen problem
  - If the probability of failure without restarts is 1/p, then an average of p restarts is needed
    - Success occurs 94% of the time, so average restarts = $\frac{1}{0.94} = 1.06$
    - 1 x 21 + .06 x 64 = 25 steps
      - (Assuming that after 2 attempts, success occurs with probability of 1)
      - 21 represents the average number of steps required for successful completion
      - Failed completion occurs with probability of .06 and requires average of 64 steps

- Completeness:
  - Probabilistically complete: As time passes, probability of being complete $\rightarrow 1$
- Stats: 3,000,000-queen problem with random-walk and restarts can be solved in minutes

1.2 Simulated Annealing

Hill climbing can not go down hill or move off local extrema without restarts, and is therefore, not complete. Random walk is complete but is not efficient. Simulated Annealing combines the advantages of both: It behaves as a random-walk in the beginning and progresses toward hill-climbing toward the end of the search procedure. Figure 3 (left) illustrates how the hill climbing portion keeps the ball moving downhill, while the random walk keeps it out of the local basins.

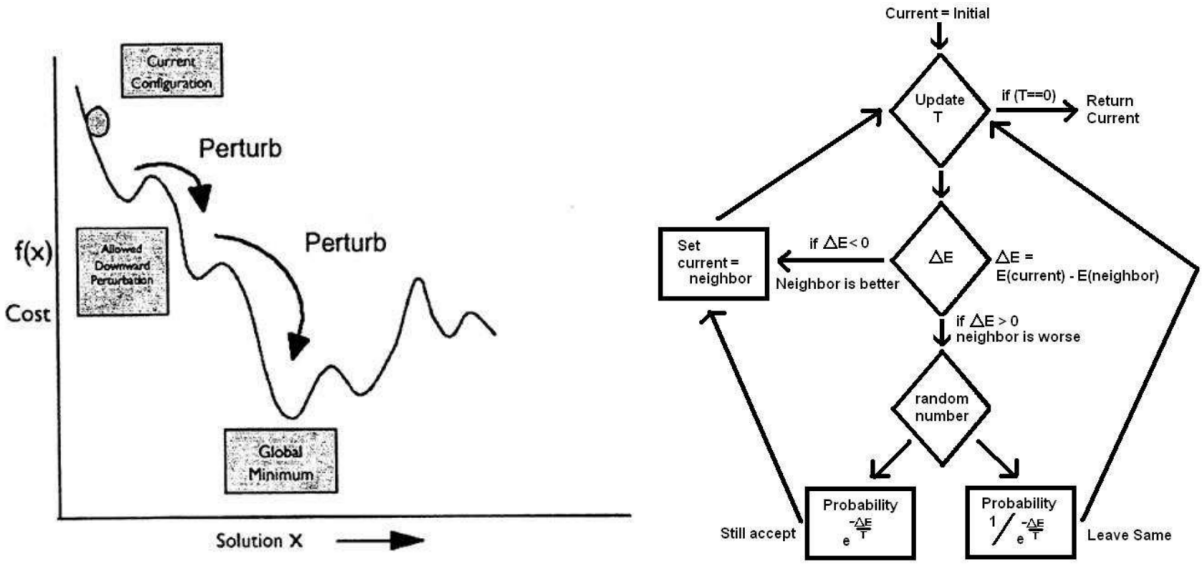

Figure 3: Simulated Annealing: The algorithm will sometimes go "uphill", against the direction of the evaluation function, if T is high enough, to prevent become stuck at local minima.

Down-hill movement is allowed with a certain probability. This probability decreases as a function of time. The resulting probability distribution has been motivated by the field of metallurgy and is described using the Boltzmann distribution: $e^{-\Delta E/T}$, where:

- ΔE = value(current) $-$ value(neighbor)
- T = Temperature
  - With high temperatures: Increased probability of going down hill (e.g. like random walk)
  - With low temperatures: Primarily hill-climbing

A flowchart showing the process of simulated annealing is shown in figure 3 (right). The degree of "randomness" in simulated annealing is based on the temperature parameter "T", which decreases by some function at the beginning of every iteration. How fast or slow the temperature decreases is defined by the so called "annealing" or "cooling schedule", which is an input parameter to the algorithm. If the temperature falls below a threshold (e.g., in the

flowchart when $T == 0$), the algorithm is complete and the current value is returned as the best solution. As long as T exceeds the threshold, the value of a neighboring location is compared against the current value (ΔE). If the neighboring value is better, the neighboring state is assigned to the current state and the loop continues. If the neighboring value is smaller, then the neighbor will be accepted depending on the Boltzman distribution defined for the current temperature value T and difference in value ΔE: $e^{-\Delta E/T}$ (bottom part of the flowchart).
Simulated annealing has the following advantages:

- Probabilistically complete: Finds optimal solution with a probability $\rightarrow 1$ (assuming temperature decreases slowly enough)
- Space complexity: O(1)

Nevertheless, it is not clear how to appropriately define the annealing scheme to solve specific problems efficiently.

1.3 Local Beam Search

Traditional local search remembers and evolves a single state. Local Beam Search attempts to better explore a search space by remembering multiple states simultaneously. The multiple states may evolve independently or there may be some exchange of information between them. The fitness of each state is compared against one another, allowing unproductive regions to be removed from consideration, while allowing resources to be committed more heavily to promising areas.

Figure 4 shows how a three-individual population evolves after one iteration. After successors have been generated from the individuals in the initial population, all values are compared and the best three continue to the next iteration. In this case, the population containing 4,-1,5 evolves to 8,6,5.

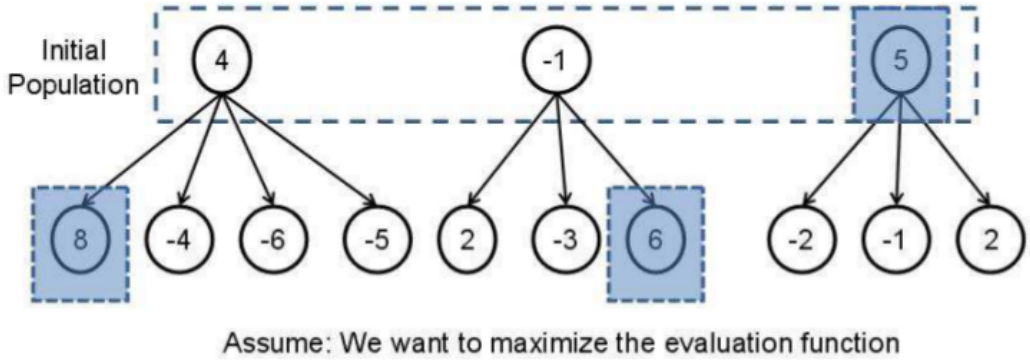

Figure 4: Local Beam Search.

- Advantage:
  - Space complexity is O(n), where "n" is the number of individuals in the population
- Disadvantage:
  - Not complete and not optimal.
  - Can get stuck in local minima.
  - Can easily degrade to simple hill climbing. Figure 5 shows a minimization example where the values of the successors of a single individual surpass all others. This causes the future population to be committed to that location, even though the other areas may still be promising. In this case, the performance of local beam search reduces to traditional hill climbing, albeit with worse space complexity.

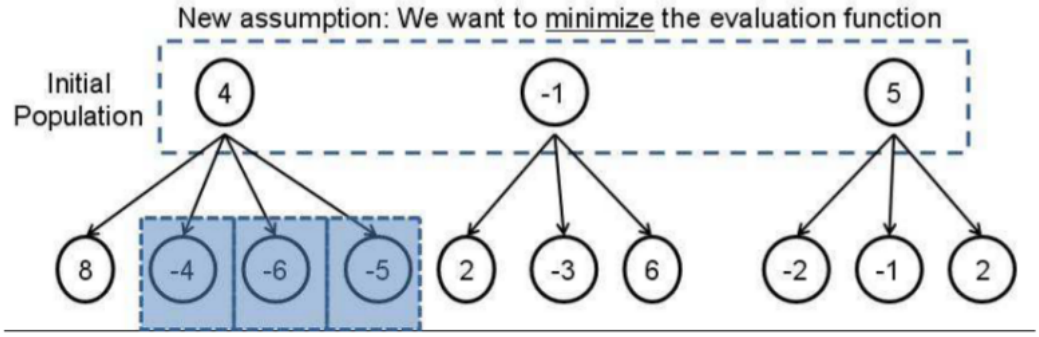

Figure 5: Local beam degrading to hill climbing.

2 Genetic Algorithms

Genetic algorithms are a variation of local-beam search, as they also store a population of individual states. The difference in genetic algorithms is that an individual is not a direct decedent of a single individual from the previous population. Instead operations are used that are inspired by biological reproduction to combine the information stored in two states to produce the new population. Typically in genetic algorithms information from the available states is encoded into strings. Given a set of such strings as the state population, a typical genetic algorithms applies the following steps:

- **Selection:** The fitness of each string is determined by a function of its performance. Strings with high fitness are provided a higher probability of staying among the population, while low-fitness strings are culled. Variability is introduced into the population using "reproduction" and "mutation".
- **Reproduction (Crossover):** Two strings are combined in a way that resembles biological genetic recombination. Zero or more crossover points are randomly assigned to the string (chromosome) and information between these crossover points are traded between individuals, yielding two new individuals. Column (c→d) in figure 6 shows a crossover event around a single crossover point. The individual in the top two rows exchange information after the third string position and individuals in the bottom two rows exchange information after the sixth string position.
- **Mutation:** Randomly selected positions on the string are changed to a new random value. Column (d→e) shows a mutation event where position 6 in the first individual, position 3 in the third individual, and position 8 in the forth individual have been changed.

Figure 7 shows how a genetic algorithm can be applied to the 8-queen problem. Each board is represented by a string of eight numbers, where each number represents the position of the queen in that column. Here, the first board corresponds to the first individual in column (c) of figure 6 and the second board corresponds to the second individual in the column. During crossover, a portion of board-one is swapped with the equal portion of board-two. Board-three results, which corresponds to the first sequence of column(d) of figure 6. Mutation would have the effect of moving a queen in one column to a new randomly selected position in that column.

- Advantages: Effective on very large search spaces. Effective on a wide range of problems
- Disadvantages: Not necessarily efficient or complete

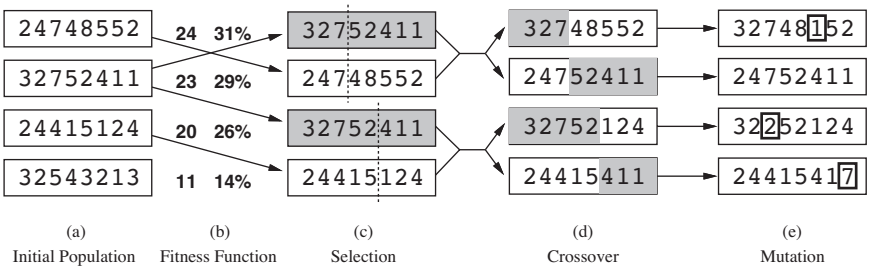

Figure 6: Genetic algorithm example.

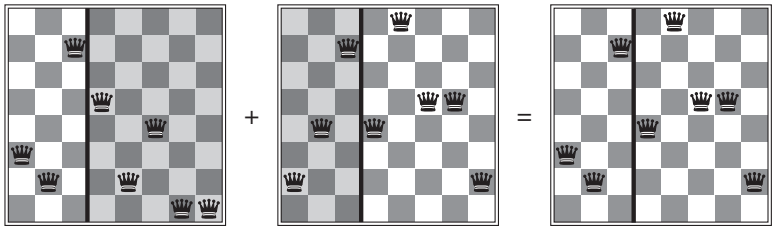

Figure 7: Crossover operation applied to 8-queen problem.

2.1 GA Example

Problem: Find with a genetic Algorithm the 5 digit binary number that has the max number of 1s (goal state is 11111), given the following initial population:
00101
11000
01101
00001

Random Numbers for Selection = 0.67,0.12,0.79,0.36 (reused consecutively)
Random Crossover Positions: 1st iteration: after 2nd bit, 2nd iteration: after 3rd bit.
Random Mutation Positions: 1st string 3rd bit, 2nd string 4th bit.

Solution:

Iteration 1

i) **Selection:** We must first compute the value of each state. Since our goal is to have all 1s lets set the evaluation function to be the number of 1s in each state, as follows:
00101 f(1) = 2
11000 f(2) = 2
01101 f(3) = 3
00001 f(4) = 1
Second step is to compute the probability of selecting a state during the selection step. The probability of selection for each node can be computed as follows: $P(n) = \frac{f(n)}{\sum_{\forall m} f(m)}$. In this way the most desired state will have the highest probability of being selected for reproduction and the sum of the probabilities will sum to 1. Note that in this example: $\sum_{\forall m} f(m) = 8 \;\; (= 2 + 2 + 3 + 1)$. Consequently the probabilities are:

(1) 00101 P(1) = 2/8 = .25
(2) 11000 P(2) = 2/8 = .25
(3) 01101 P(3) = 3/8 = .375
(4) 00001 P(4) = 1/8 = .125

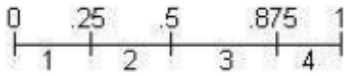

Figure 8: Normalization of population.

In order to actually select the states we apply the following approach. We will use a random number generator that returns numbers in the range 0 to 1. Depending on the number that the generator returns, we want to select a state according to the probability distribution given above. If the random number is in the range [0 , P(1)] then the first state will be selected. For the following states, we accumulate the probabilities to compute the corresponding range. In order for state i to be selected, the random number must be in the range: $[\sum_{\forall j<i} P(j), \sum_{\forall j \leq i} P(j)]$. Figure 8 provides the exact ranges that correspond to the probabilities of the four strings we have available.

Given the specified random numbers, the following states will be selected for reproduction:
(previously state # 3) 01101
(previously state # 1) 00101
(previously state # 3) 01101
(previously state # 2) 11000

Note how the algorithm promoted the selection of the best performing state (state 3), resulting in it being appearing twice in the resulting population.

ii) **Crossover:** We now swap the bits on a random bit location. In this example the first swap occurs after the 2nd bit.

(3) 01101 to 01101
(1) 00101 to 00101
(3) 01101 to 01000
(2) 11000 to 11101

iii) **Mutation:** Now we add a mutation, which is to flip a randomly selected single bit of the string. In this case, the mutation will happen on the 1st string at bit 3 and on the 2nd string at bit 4.

01101 to 01001
00101 to 00111
01000
11101

Iteration 2

We repeat the same process while using the same random values. This time the crossover position is at bit 3 and mutations will occur on the 1st string at bit 3 and on the 2nd string at bit 4.

i) **Selection:** The probability distribution will look as follows:

(1) 01001 P(1) = 2/ (2+3+1+4 = 10) = .2
(2) 00111 P(2) = 3/10 = .3
(3) 01000 P(3) = 1/10 = .1
(4) 11101 P(4) = 4/10 = .4

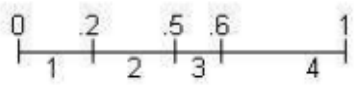

Figure 9: Normalization of population.

Using probabilistic selection, the new population is:
(previously state # 4) 11101
(previously state # 1) 01001
(previously state # 4) 11101
(previously state # 2) 00111

ii) **Crossover:** After crossover at 3rd bit:
(4) 11101 to 11001
(1) 01001 to 01101
(4) 11111
(2) 00101

iii) **Mutation:** After mutation on the 1st string at bit 3 and 2nd string at bit 4:
(4) 11001 to 11101
(1) 01101 to 01111
(4) 11111 (goal state reached)
(2) 00101