Artificial Intelligence and Machine Learning (AIML)

2023–24

- **Last lecture**: dynamic programming
- **This lecture**: approximate methods for combinatorial optimization in AI, greedy approximation methods

AIML

Approximate methods: overview

- Exact optimization methods are completely reliable, but typically quite slow, approximate methods are fast but not necessarily reliable
- Approximate methods cannot guarantee finding an optimal solution, only one which is 'good enough' N.B. -> “Nota bene” means “note well”
- Various strategies for approximate optimization: **stochastic** (randomized), **greedy** (N.B. not exact greedy!), **tabu**, **genetic optimization**, **particle swarm optimization**, **simulated annealing**, **local beam search**
- Approximate methods aim to find a configuration which is at least better than the starting configuration

Approximate methods: overview

- Exact optimization methods are completely reliable, but typically quite slow, approximate methods are fast but not necessarily reliable
- Approximate methods cannot guarantee finding an optimal solution, only one which is 'good enough’

Approximate methods: overview

- Exact optimization methods are completely reliable, but typically quite slow, approximate methods are fast but not necessarily reliable
- Approximate methods cannot guarantee finding an optimal solution, only one which is 'good enough'
- Various strategies for approximate optimization: **stochastic** (randomized), **greedy** (N.B. not exact greedy!), **tabu**, **genetic optimization**, **particle swarm optimization**, **simulated annealing**, **local beam search**

Approximate methods: overview

- Exact optimization methods are completely reliable, but typically quite slow, approximate methods are fast but not necessarily reliable
- Approximate methods cannot guarantee finding an optimal solution, only one which is 'good enough'
- Various strategies for approximate optimization: **stochastic** (randomized), **greedy** (N.B. not exact greedy!), **tabu**, **genetic optimization**, **particle swarm optimization**, **simulated annealing**, **local beam search**
- Approximate methods aim to find a configuration which is at least better than the starting configuration

AIML

Approximate methods: analysis

- Some stochastic algorithms have optimal convergence guarantees, example: **Markov chain Monte Carlo** (**MCMC**), but only in the limit of an infinite number of iterations

Approximate methods: analysis

- Some stochastic algorithms have optimal convergence guarantees, example: **Markov chain Monte Carlo** (**MCMC**), but only in the limit of an infinite number of iterations
- Even the number of iterations required to find a configuration whose objective is within some required tolerance of the optimal solution, is typically not known

Approximate methods: analysis

- Some stochastic algorithms have optimal convergence guarantees, example: **Markov chain Monte Carlo** (**MCMC**), but only in the limit of an infinite number of iterations
- Even the number of iterations required to find a configuration whose objective is within some required tolerance of the optimal solution, is typically not known
- Otherwise, no guarantee that a longer search leads to a better solution; most approximate methods can get trapped in local optima

AIML

Approximate greedy: formulation

- Also known as **hill-climbing** or **iterative improvement**, defines a **combinatorial neighbourhood** $\mathcal{N}(X)$ of a given configuration X, configurations which are somehow 'close' to X under some distance **metric**

度量

Approximate greedy: formulation

- Also known as **hill-climbing** or **iterative improvement**, defines a **combinatorial neighbourhood** $\mathcal{N}(X)$ of a given configuration X, configurations which are somehow 'close' to X under some distance **metric**
- Example metric: **Hamming distance** $d(X, X')$, number of elements of the configuration X which differ from configuration X'

Approximate greedy: formulation

- Also known as **hill-climbing** or **iterative improvement**, defines a **combinatorial neighbourhood** $\mathcal{N}(X)$ of a given configuration X, configurations which are somehow 'close' to X under some distance **metric**
- Example metric: **Hamming distance** $d(X, X')$, number of elements of the configuration X which differ from configuration X'
- **Idea**: from an initial bad configuration, if we repeatedly select a better one within the neighbourhood of the current configuration, we can eventually find a good one

AIML

Approximate greedy: algorithm

- **Step 1**. *Initialization*: Start with iteration $n=0$, select an initial candidate configuration X_0, choose a maximum number of iterations R.
- **Step 2**. *Neighbourhood search*: Find the configuration X' within the configuration neighbourhood $\mathcal{N}(X_n)$ with smallest objective function value $F(X')$, and set $X_{n+1} = X'$. If $F(X') \geq F(X_n)$, then terminate with $X^* = X_n$.
- **Step 3**. *Iteration*: Set $n=n+1$, and while $n \leq R$, go back to step 2, otherwise exit with solution $X^* = X_n$.

Hill-Climbing

Hill-Climbing (assuming maximisation)

1. current_solution = generate initial candidate solution randomly

2. Repeat:
    2.1 generate neighbour solutions (differ from current solution by a single element)
    2.2 best_neighbour = get highest quality neighbour of current_solution
    2.3 If quality(best_neighbour) <= quality(current_solution)
        2.3.1 Return current_solution
    2.4 current_solution = best_neighbour

Designing Representation, Initialisation and Neighbourhood Operators

Hill-Climbing (assuming maximisation)

1. current_solution = generate initial solution randomly

2. Repeat:

    2.1 generate neighbour solutions (differ from current solution by a single element)

    2.2 best_neighbour = get highest quality neighbour of current_solution

    2.3 If quality(best_neighbour) <= quality(current_solution)

        2.3.1 Return current_solution

    2.4 current_solution = best_neighbour

- Representation:
  - How to store the design variable.
  - E.g., boolean, integer or float variable or array.
- Initialisation procedure:
  - Usually involve randomness.
- Neighbourhood operator:
  - How to generate neighbour solutions.

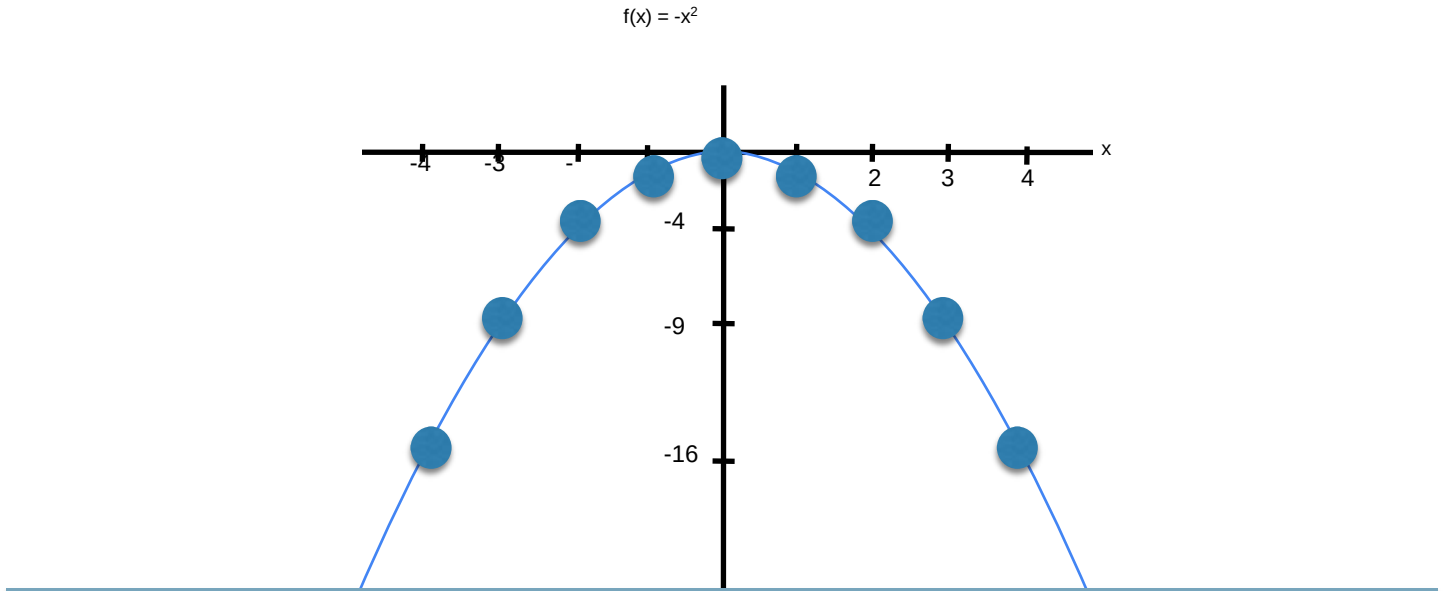
Illustrative Example

- Design variables represent a candidate solution.
  - $x \in Z$
  - Our search space are all integer numbers.
- [Optional] Solutions must satisfy certain constraints, which define solution feasibility.
  - None
- Objective function defines the quality (or cost) of a solution.
  - $f(x) = -x^2$, to be maximised

Illustrative Example

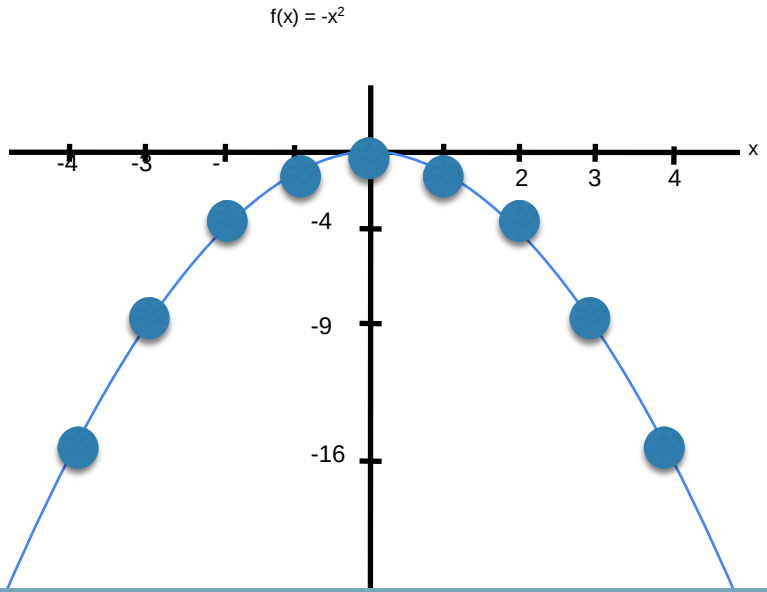
- Representation:
  - Integer variable.
- Initialisation procedure:
  - Initialise with an integer value picked uniformly at random.
- Neighbourhood operator:
  - Add or subtract 1.

Illustrative Example



This is an illustrative example to understand the behaviour of the algorithm. In reality, we are unlikely to know the shape of our function to be optimised beforehand.

Illustrative Example



Other problems may have more dimensions,
and more neighbours for each candidate solution.

Illustrative Example

Hill-Climbing (assuming maximisation)

1. current_solution = generate initial solution randomly

2. Repeat:

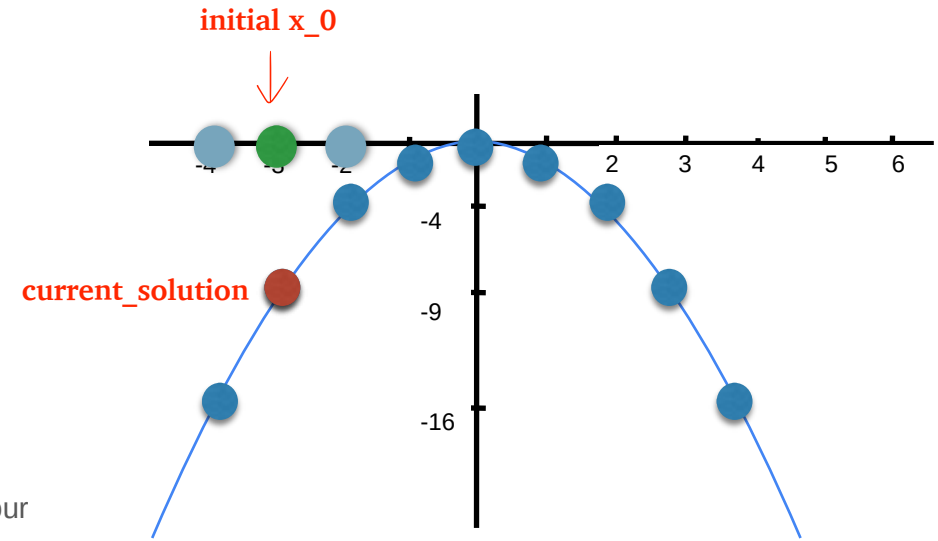
    2.1 generate neighbour solutions (differ from current solution by a single element)

    2.2 best_neighbour = get highest quality neighbour of current_solution

    2.3 If quality(best_neighbour) <= quality(current_solution)

        2.3.1 Return current_solution

    2.4 current_solution = best_neighbour



Illustrative Example

Hill-Climbing (assuming maximisation)

1. current_solution = generate initial solution randomly

2. Repeat:

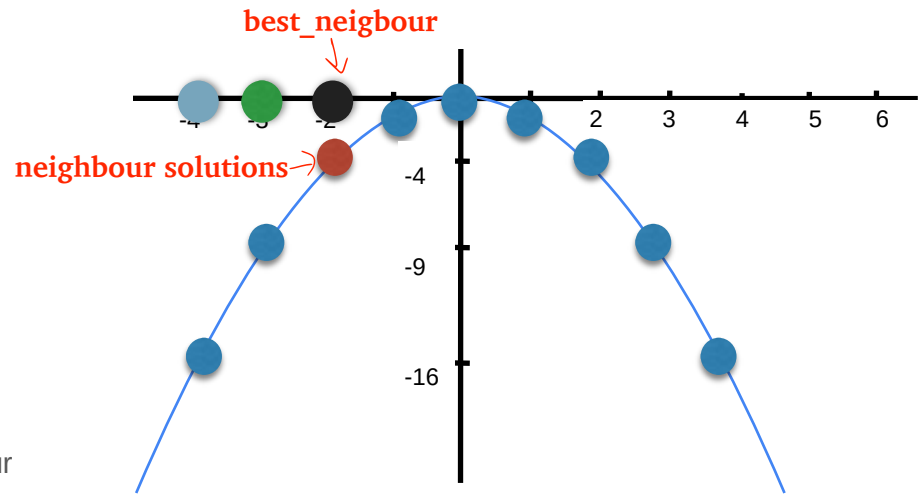
    2.1 generate neighbour solutions (differ from current solution by a single element)

    2.2 best_neighbour = get highest quality neighbour of current_solution

    2.3 If quality(best_neighbour) <= quality(current_solution)

        2.3.1 Return current_solution

    2.4 current_solution = best_neighbour



Illustrative Example

Hill-Climbing (assuming maximisation)

1. current_solution = generate initial solution randomly

2. Repeat:

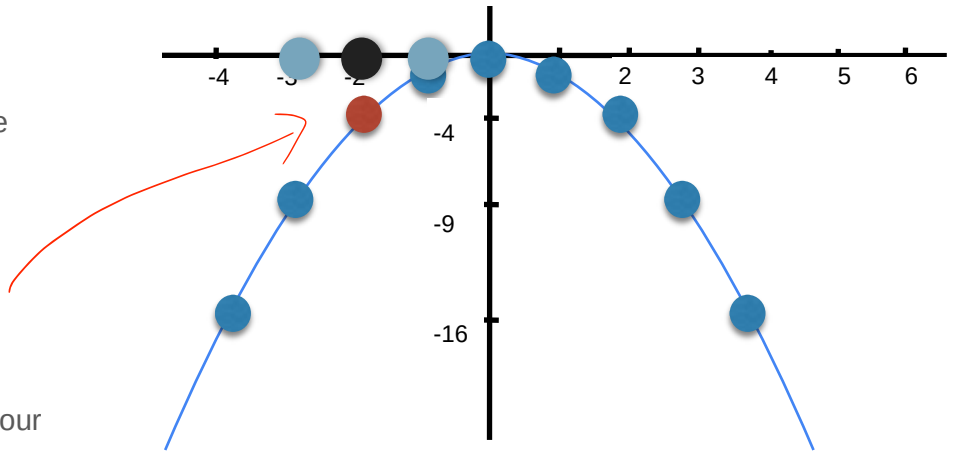
    2.1 generate neighbour solutions (differ from current solution by a single element)

    2.2 best_neighbour = get highest quality neighbour of current_solution

    2.3 If quality(best_neighbour) <= quality(current_solution)

        2.3.1 Return current_solution

    2.4 current_solution = best_neighbour



Illustrative Example

Hill-Climbing (assuming maximisation)

1. current_solution = generate initial solution randomly

2. Repeat:

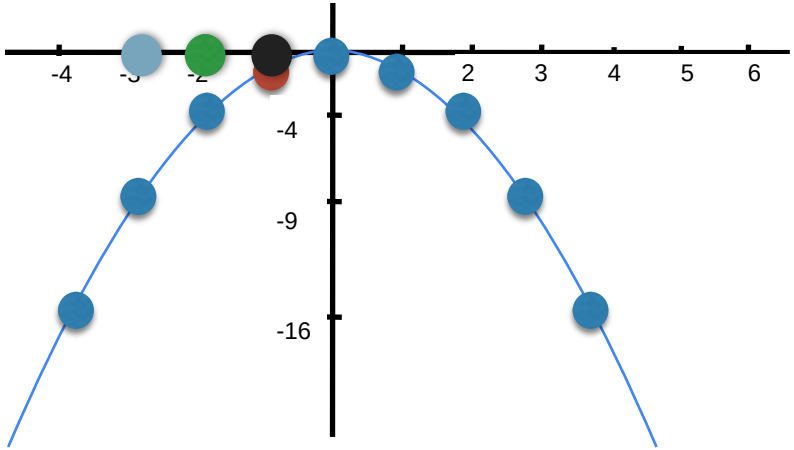
    2.1 generate neighbour solutions (differ from current solution by a single element)

    2.2 best_neighbour = get highest quality neighbour of current_solution

    2.3 If quality(best_neighbour) <= quality(current_solution)

        2.3.1 Return current_solution

    2.4 current_solution = best_neighbour



Illustrative Example

Hill-Climbing (assuming maximisation)

1. current_solution = generate initial solution randomly

2. Repeat:

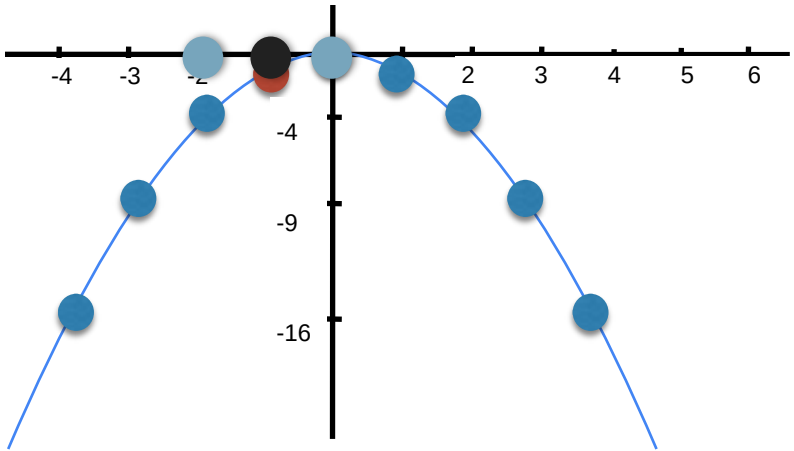
    2.1 generate neighbour solutions (differ from current solution by a single element)

    2.2 best_neighbour = get highest quality neighbour of current_solution

    2.3 If quality(best_neighbour) <= quality(current_solution)

        2.3.1 Return current_solution

    2.4 current_solution = best_neighbour



Illustrative Example

Hill-Climbing (assuming maximisation)

1. current_solution = generate initial solution randomly

2. Repeat:

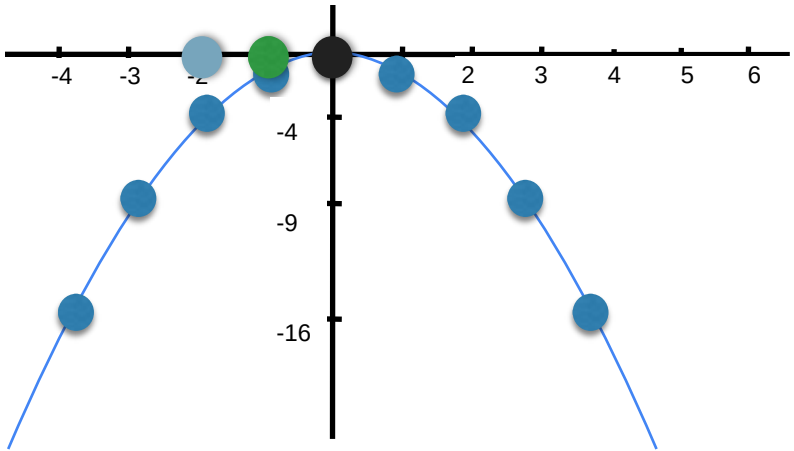
    2.1 generate neighbour solutions (differ from current solution by a single element)

    2.2 best_neighbour = get highest quality neighbour of current_solution

    2.3 If quality(best_neighbour) <= quality(current_solution)

        2.3.1 Return current_solution

    2.4 current_solution = best_neighbour



Illustrative Example

Hill-Climbing (assuming maximisation)

1. current_solution = generate initial solution randomly

2. Repeat:

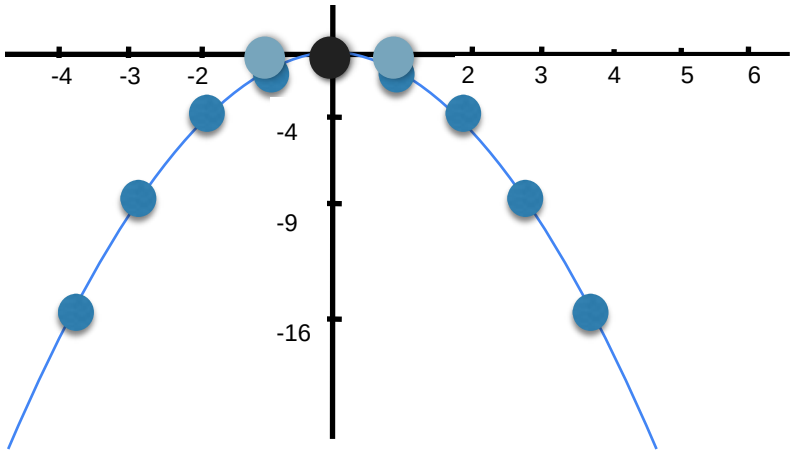
    2.1 generate neighbour solutions (differ from current solution by a single element)

    2.2 best_neighbour = get highest quality neighbour of current_solution

    2.3 If quality(best_neighbour) <= quality(current_solution)

        2.3.1 Return current_solution

    2.4 current_solution = best_neighbour



Illustrative Example

Hill-Climbing (assuming maximisation)

1. current_solution = generate initial solution randomly

2. Repeat:

    2.1 generate neighbour solutions (differ from current solution by a single element)

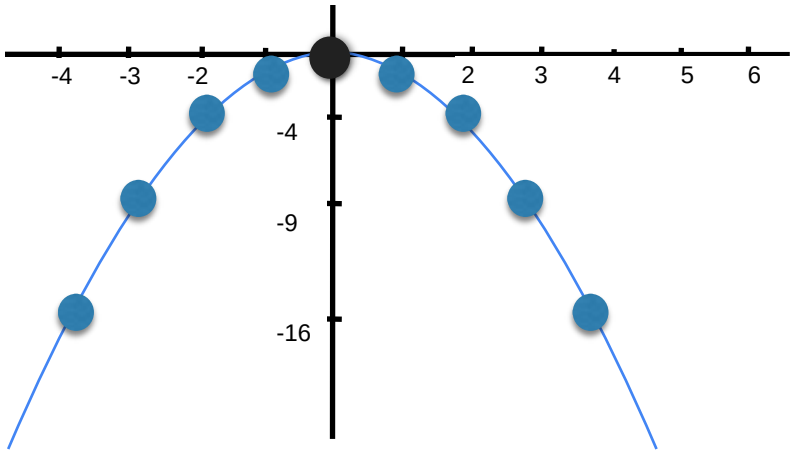
    2.2 best_neighbour = get highest quality neighbour of current_solution

    2.3 If quality(best_neighbour) <= quality(current_solution)

        2.3.1 Return current_solution

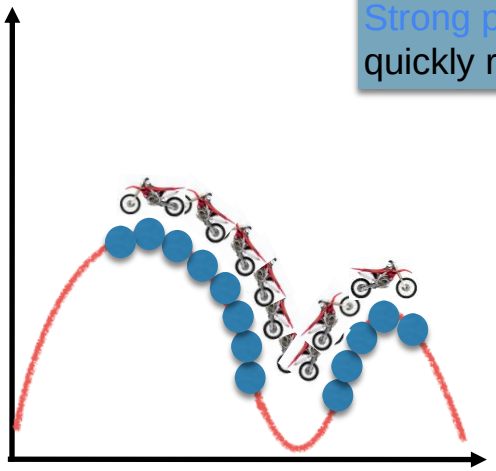
    2.4 current_solution = best_neighbour

Until a maximum number of iterations



General Idea

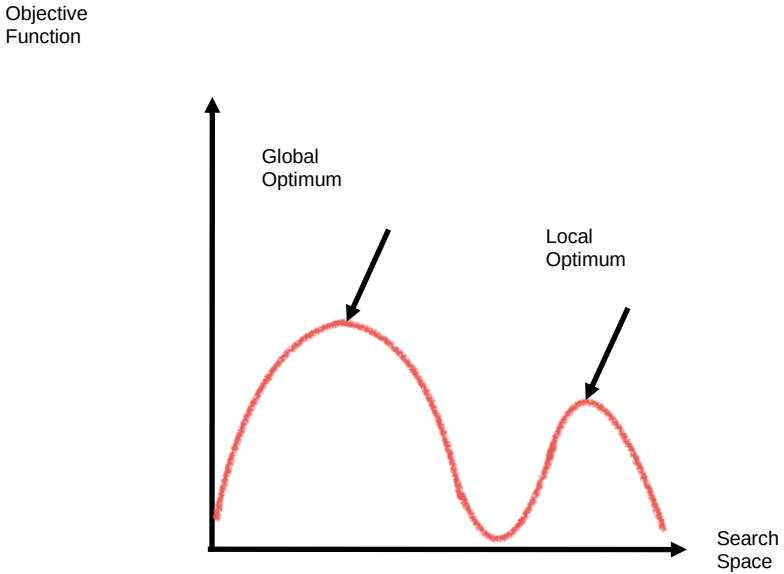
Objective Function



Search Space

Strong point: Hill climbing allows you to quickly reach the top.

Greedy Local Search

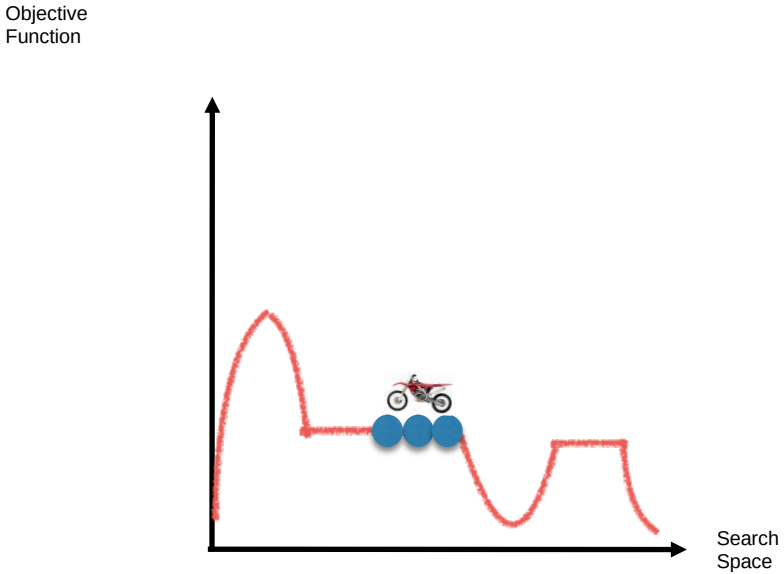


Weakness: Hill-climbing may get trapped in a local optimum.

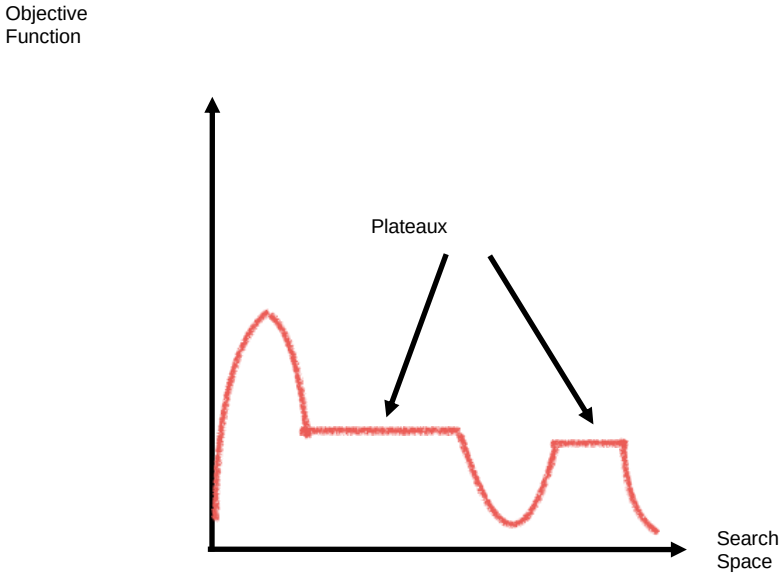
Hill-climbing is a local search method.

Hill-climbing is greedy.

Greedy Local Search



Greedy Local Search



Weakness: Hill-climbing may get trapped in plateaus.

Optimality, Time and Space Complexity

- Optimality:
  - Hill Climbing is not guaranteed to find optimal solutions.
- Time complexity (worst case scenario):
  - We will run until the maximum number of iterations *m* is reached.
  - Within each iteration, we will generate a maximum number of neighbours *n*, each of which may take O(p) each to generate.
  - Worst case scenario: O(mnp).
- Space complexity (worst case scenario):
  - Assume that the design variable is represented by O(q).
  - Within each iteration, we will generate a maximum number of neighbours *n*.
  - Assume that neighbours are generated sequentially and that the space *r* needed to generate them is negligible compared to *n* and *q.*
  - Space complexity: O(nq+r) = O(nq).

Example 2: maximum sum combination

- **Problem specification**: N input data items, combination size M

$$X^\star = \arg\max_{X' \in \mathcal{X}} \sum_{x' \in X'} x'$$

Approximate greedy: maximum sum combination

- **Problem specification**: N input data items, combination size M
- **Objective function**: $F(X) = \sum_{x' \in X} x'$, configurations are size-M subsets of x.
- **Optimization problem**:

$$X^\star = \arg\max_{X' \in \mathcal{X}} \sum_{x' \in X'} x'$$

Approximate greedy: maximum sum combination

- **Problem specification**: N input data items, combination size M
- **Objective function**: $F(X) = \sum_{x' \in X} x'$, configurations are size-M subsets of x.

Summation of x' configurations that belongs to X.

- **Optimization problem**:

$$X^\star = \arg\max_{X' \in \mathcal{X}} \sum_{x' \in X'} x'$$

- Use greedy approximate search with Hamming distance neighbourhood size r

Approximate greedy: maximum sum combination

- **Problem specification**: N input data items, combination size M
- **Objective function**: $F(X) = \sum_{x' \in X} x'$, configurations are size-M subsets of x.
- **Optimization problem**:

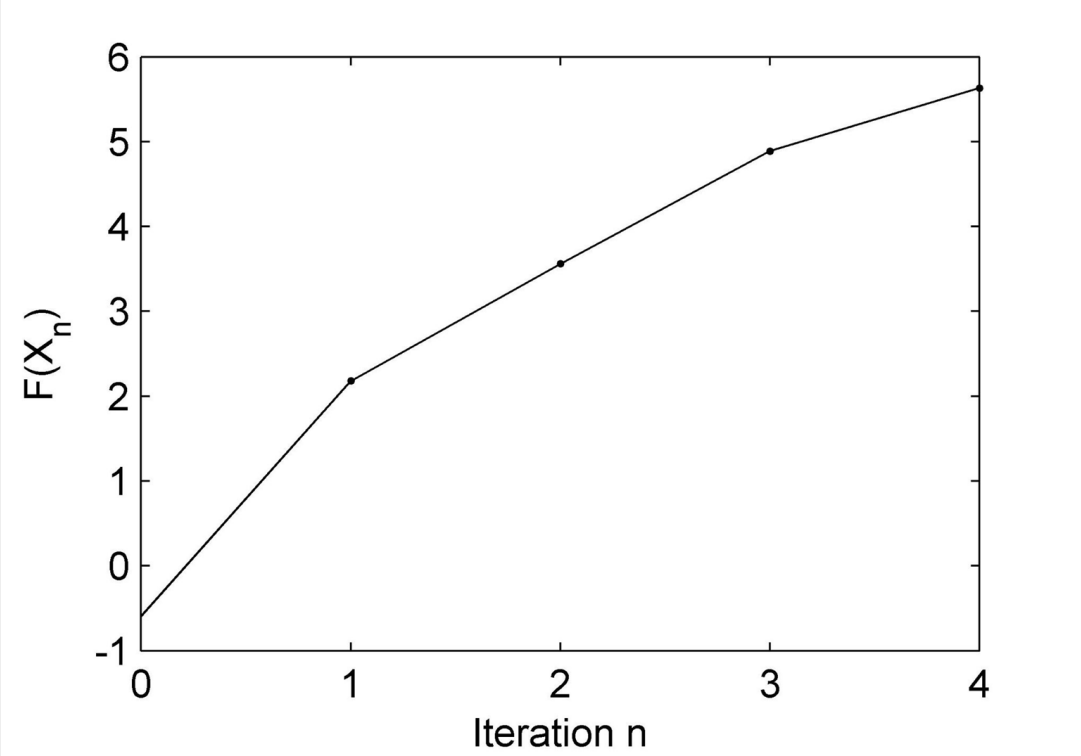
$$X^\star = \arg\max_{X' \in \mathcal{X}} \sum_{x' \in X'} x'$$

- Use greedy approximate search with Hamming distance neighbourhood size r
- **Example**: filling magazine advertising slots at maximum profit, absorbing loss leaders

Example:

Input data: x=[-0.43, -1.67, 0.13, 0.29, -1.15, 1.19, 1.19, -0.04, 0.33, 0.17, -0.19, 0.73, -0.59, 2.18, -0.14, 0.11, 1.07, 0.06, -0.10, -0.83]
Globally optimal solution: X^*={1.19,1.19,2.18,1.07}, $F(X^*)$=5.63, possible combinations: 4845

Iteration	**Configuration** X_n	**Objective value** $F(X_n)$
n=0 (initialize)	{0.33,-0.19,-0.59,-0.14}	-0.59
n=1	{0.33,-0.19,**2.18**,-0.14}	2.18
n=2	{0.33,**1.19**,2.18,-0.14}	3.56
n=3	{**1.19**,1.19,0.33,2.18}	4.89
n=4 (terminate)	{1.19,1.19,**1.07**,2.18}	5.63


6
5
4
3
2
1
0
-1
F(X_n)
0
1
2
3
4
Iteration n

Approximate greedy: analysis

- The sequence of candidate configurations will have decreasing objective function value, i.e. $F(X_{n+1}) < F(X_n)$ for all $n=0,1,\ldots,R$

Approximate greedy: analysis

- The sequence of candidate configurations will have decreasing objective function value, i.e. $F(X_{n+1}) < F(X_n)$ for all $n=0,1,\ldots,R$
- Final configuration is not guaranteed to be optimal.

Approximate greedy: analysis

- The sequence of candidate configurations will have decreasing objective function value, i.e. $F(X_{n+1}) < F(X_n)$ for all $n=0,1,\ldots,R$
- Final configuration is not guaranteed to be optimal.
- We cannot predict in advance when it will terminate, except that it will not exceed R iterations.

Approximate greedy: analysis

- The sequence of candidate configurations will have decreasing objective function value, i.e. $F(X_{n+1}) < F(X_n)$ for all $n=0,1,...,R$
- Final configuration is not guaranteed to be optimal.
- We cannot predict in advance when it will terminate, except that it will not exceed R iterations.
- Large local neighbourhood leads to slow search, but more chance of hitting the global optima.

References and further reading

- **R&N**, Section 4.1
- **MLSP**, Section 2.6