CO3091 - Computational Intelligence and Software Engineering

Lecture 02

Photo from: http://cdn.motocross.transworld.net/files/2009/11/img-7532.jpg

Hill-Climbing

Leandro L. Minku

Overview

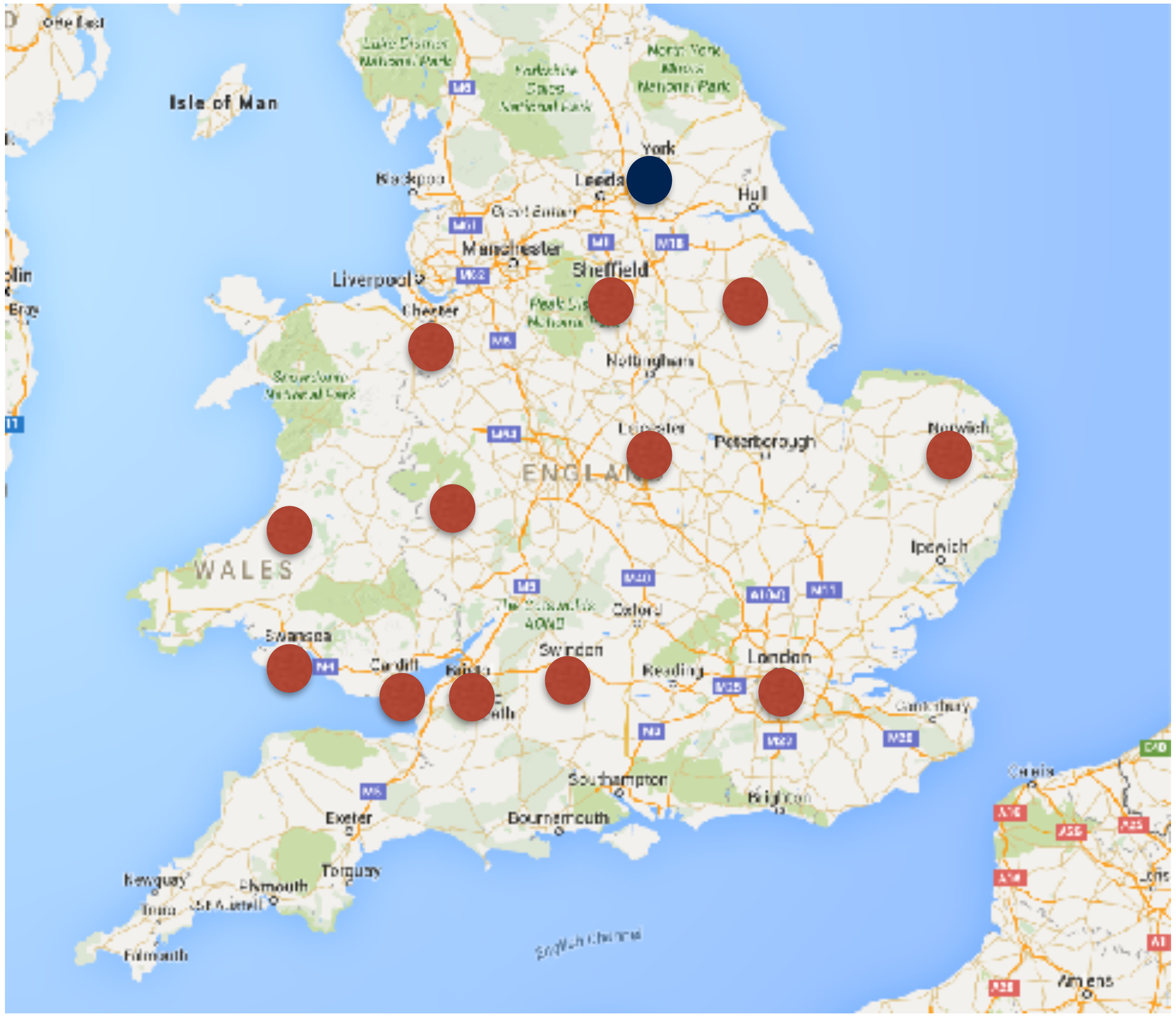
- Optimisation Problems
- Formulating Optimisation Problems
- Brute-Force Search
- Hill-Climbing
- Illustrative Example
- Example of Hill-Climbing for Software Module Clustering

Optimisation Problems

- Optimisation problems: to find a solution that achieves one or more pre-defined goals.
- Maximisation / minimisation problems.

Examples of Optimisation Problems

- Traveling Salesman Problem:
  - A salesman must travel passing through *N* cities.
  - Each city must be visited once.
  - He/she must finish where he/she was at first.
  - The path between each pair of cities has a distance (or cost).



Problem: find a sequence of cities that minimises traveling distance (or cost).

Examples of Optimisation Problems

- Bin packing problem:
  - Given bins with maximum volume *V*, which cannot be exceeded.
  - We have *n* items to pack, each with a volume *v*.
  - We must pack all items.

Problem: find an assignment of items to bins that minimises the number of bins used.

Photo from: http://maritime-connector.com/images/container-ship-16-wiki-19057.jpg

Photo from: http://www.tscargo.ca/images/cargo1.jpg

Example of Software Engineering Optimisation Problems

- Software Module Clustering:
    - Software is composed of several units, which can be organised into modules.
    - Well modularised software is easier to develop and maintain.
    - As software evolves, modularisation tends to degrade.

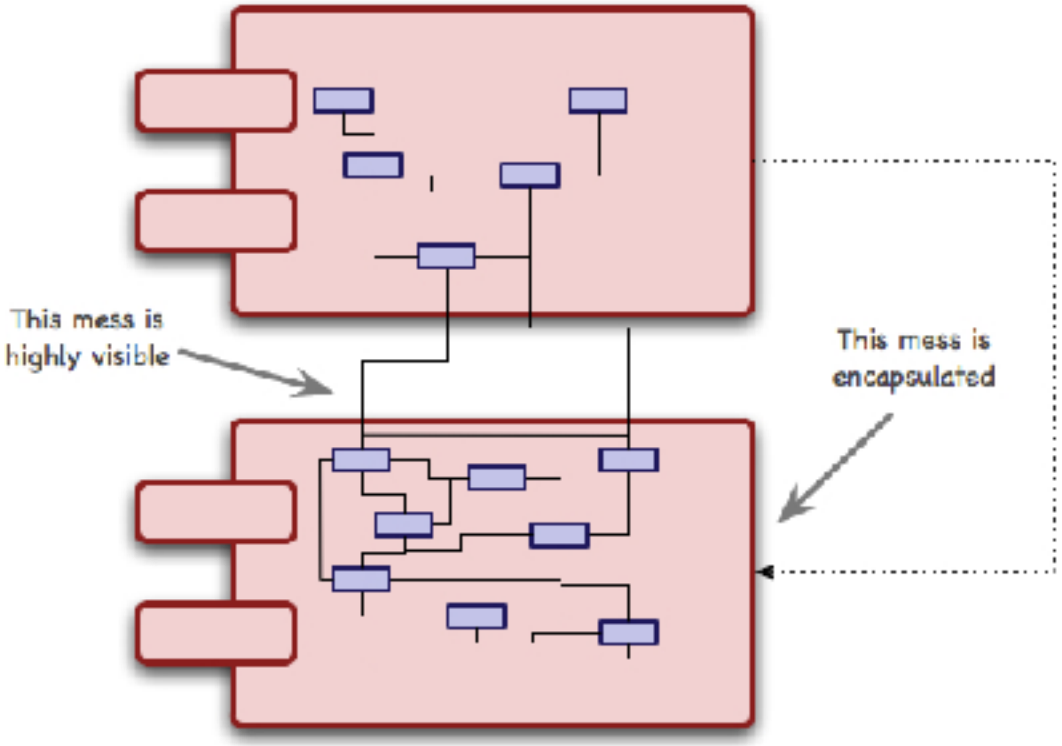


Image from: http://www.kirkk.com/modularity/wp-content/uploads/2009/12/EncapsulatingDesign1.jpg

Problem: find a grouping of units into modules that maximises the quality of modularisation.

Formulating Optimisation Problems

- Design variables represent a solution.
- Design variables define the search space of candidate solutions.
- [Optional] Solutions must satisfy certain constraints.
- Objective function defines our goal.
    - Can be used to evaluate the quality of solutions.
    - Function to be optimised (maximised or minimised).

Traveling Salesman Problem Formulation

- Design variables represent a solution.
  - Vector **x** of size *N*, where *N* is the number of cities.
  - **x** represents a sequence of cities to be visited.

- Design variables define the search space of candidate solutions.
  - All possible sequences of cities, where each city appears only once.

- [Optional] Solutions must satisfy certain constraints.
  - Each city must appear once and only once in **x**.
  - Salesman must return to the city of origin.

- Objective function defines our goal.
  - Total_distance(**x**) =
    sum of distances between consecutive cities in **x** + distance from last city to the origin.
  - To be minimised.

Brute-Force Search

- Brute-force search = exhaustive search = generate and test.
- Systematically enumerate all possible candidates for the solution and check which one is the best.
- Guaranteed to find the optimal solution.
- Can we use brute-force search to solve optimisation problems?


Image from: http://vignette2.wikia.nocookie.net/creepypasta/images/8/80/Creepycemetery.jpg/revision/latest?cb=20131013135547

Problem: high computational complexity.

Brute-Force for Traveling Salesman Problem

A solution is a sequence of cities, where each city appears only once.

- Number of cities N = 2

A	B
B	A

- Number of cities N = 3

A	B	C
A	C	B
B	A	C
B	C	A
C	A	B
C	B	A

Our sequences of cities are permutations.

Number of permutations is factorial: N!

Brute-Force for Traveling Salesman Problem

- Factorial time complexity:
    - 2! = 2
    - 3! = 6
    - …
    - 10! = 3,628,800
    - 20! = 2,432,902,008,176,640,000 ≈ 2.43 x 10^{18}

- Assume that 10^9 permutations take one second.
    - 2!/10^9 = 0.000000002s
    - 3!/10^9 = 0.000000006s
    - …
    - 10!/10^9 = 0.0036288s
    - 20!/10^9 ≈ 2,432,902,008s ≈ 77years

Brute-force works only for very small problems.

Solving Optimisation Problems Using Computational Intelligence

- Heuristic algorithms, which aim to find good solutions to problems in a reasonable amount of time.
  - Make informed guesses to guide the search about the direction to a goal.
  - Typically not guaranteed to find the optimum, but able to find sufficiently good or near-optimal solutions.
- Good for:
  - Large problems, where we cannot afford enumerating all possible solutions to guarantee optimality.
  - Problems where no exact optimisation algorithm exists that can solve the problem in polynomial time.
  - Problems where sufficiently good or near-optimal solutions are acceptable.

Hill-Climbing

Hill-Climbing (assuming maximisation)

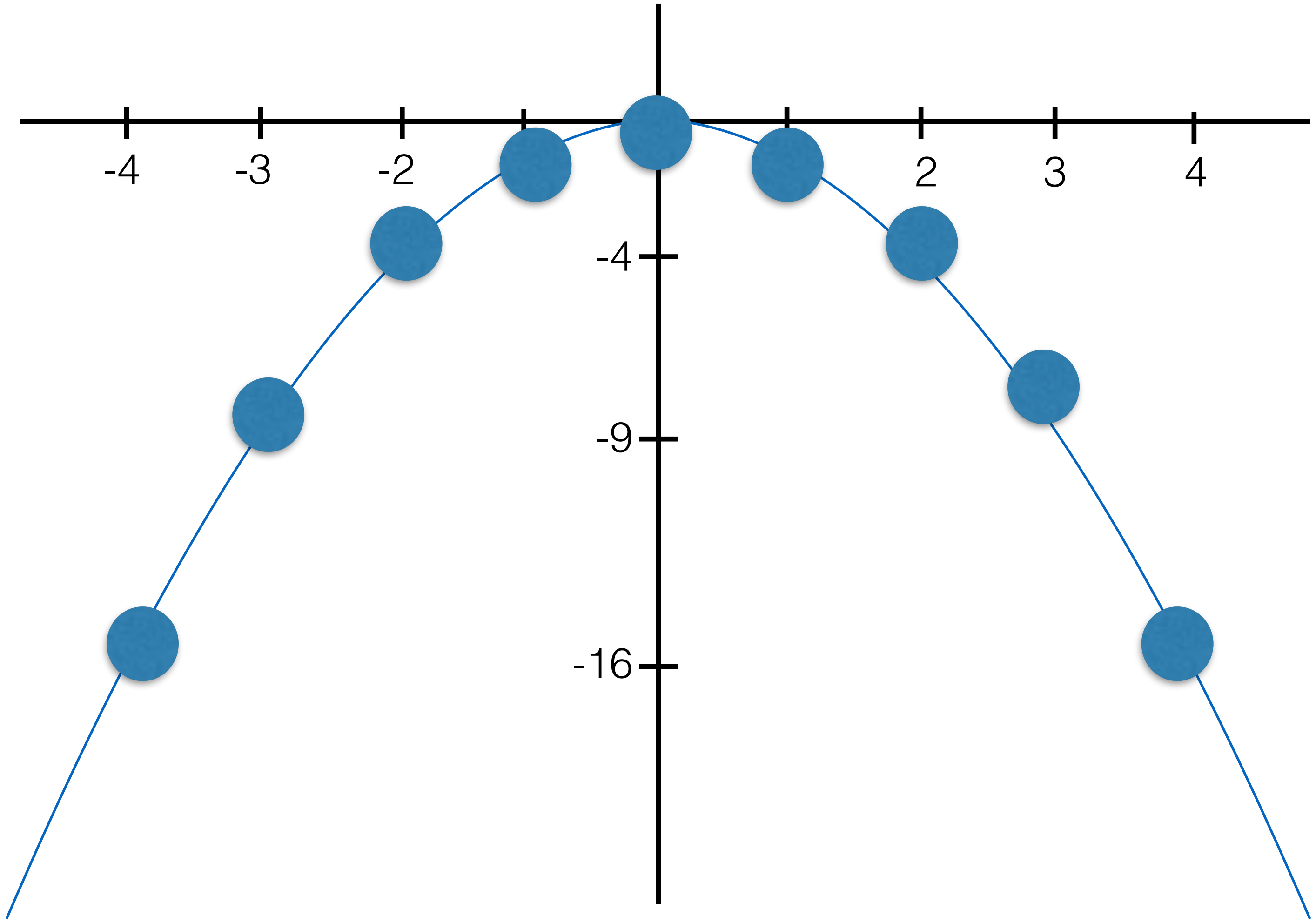
```
1. current_solution = generate initial solution randomly

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

- Design variables represent a solution.
  - **$x \in Z$**
- Design variables define the search space of candidate solutions.
  - **Our search space are all integer numbers.**
  - This also defines our neighbourhood.
- Objective function defines our goal and represents the quality of a solution.
  - Can be used to evaluate the quality of solutions.
  - Function to be optimised (maximised or minimised).
  - **$f(x) = -x^2$, to be maximised**
- [Optional] Solutions must satisfy certain constraints.
  - **None**

Illustrative Example

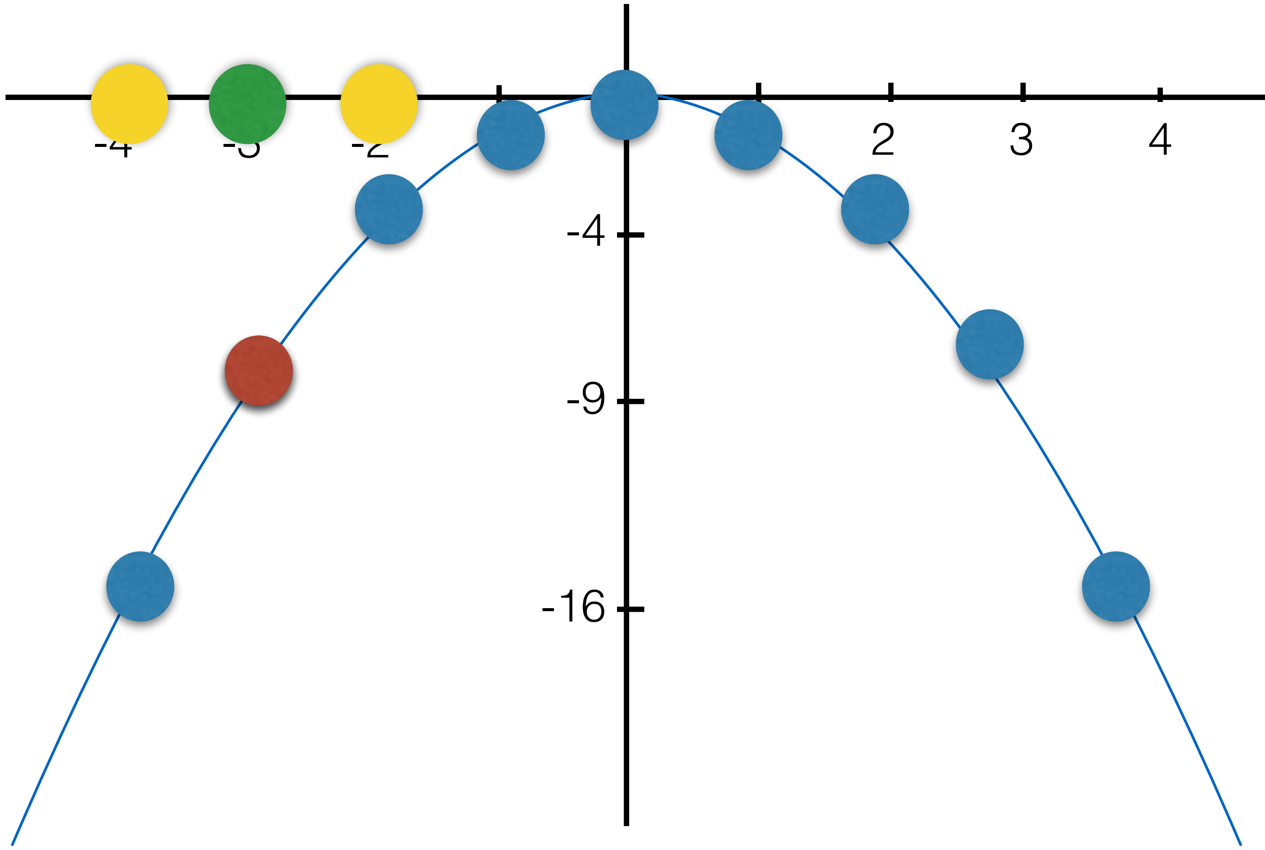
$f(x) = -x^2$



Illustrative Example

Hill-Climbing (assuming maximisation)

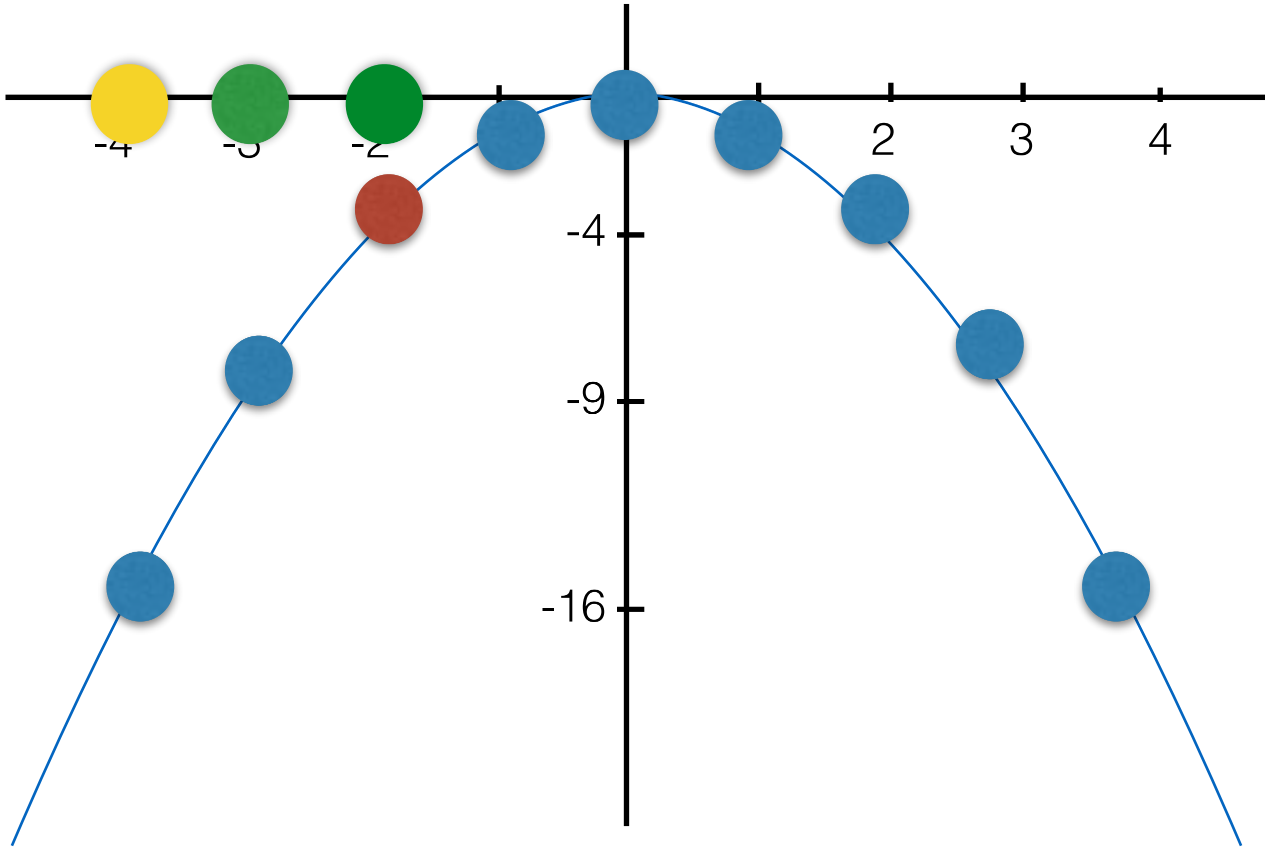
```
1. current_solution = generate initial solution randomly

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)

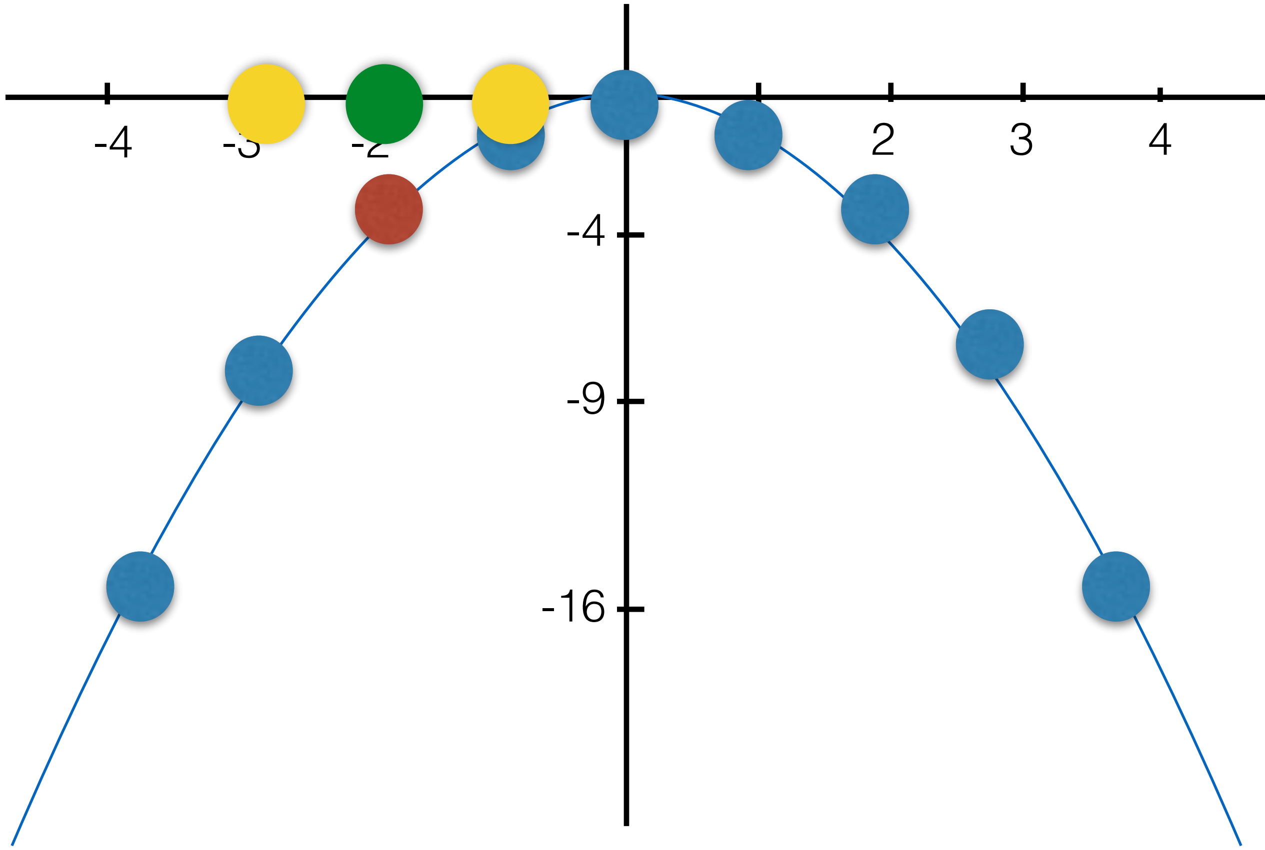
```
1. current_solution = generate initial solution randomly

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)

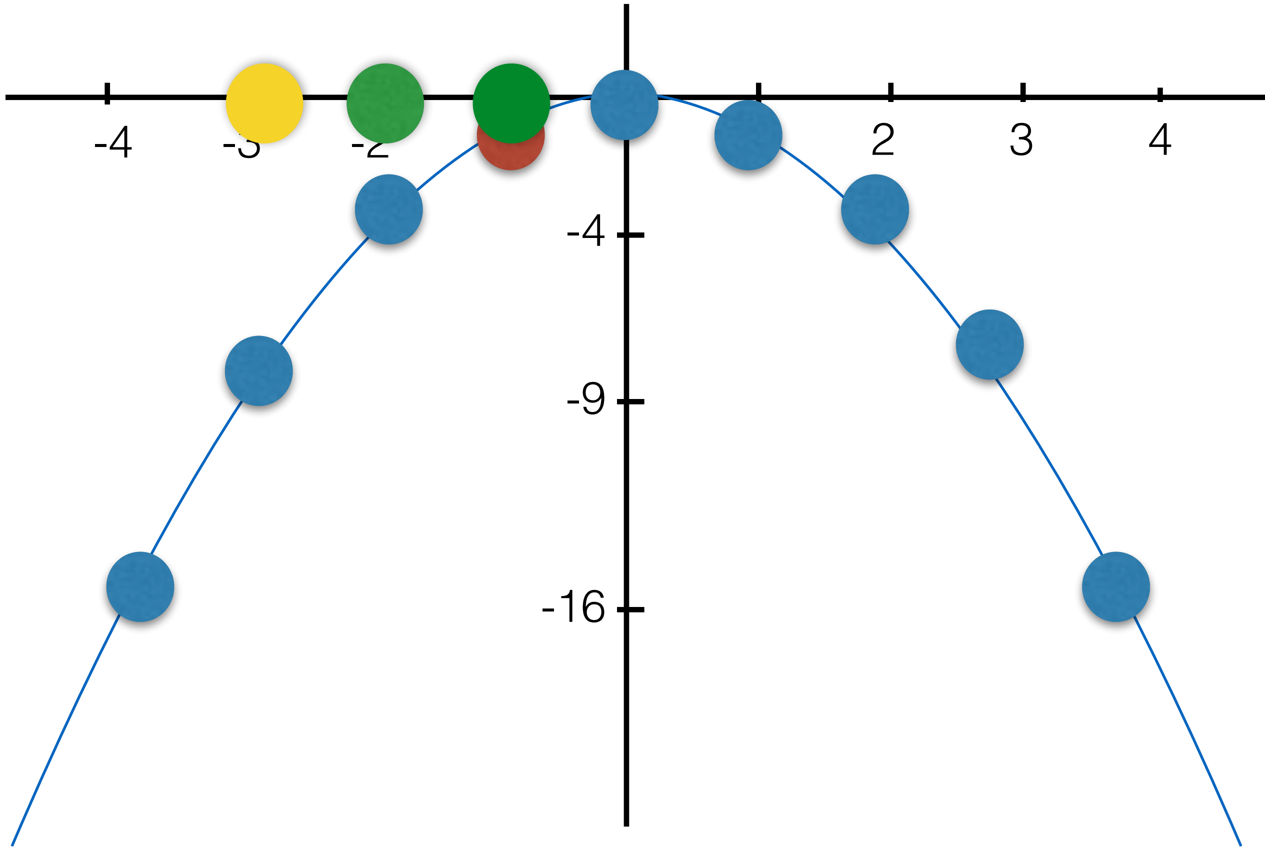
```
1. current_solution = generate initial solution randomly

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)

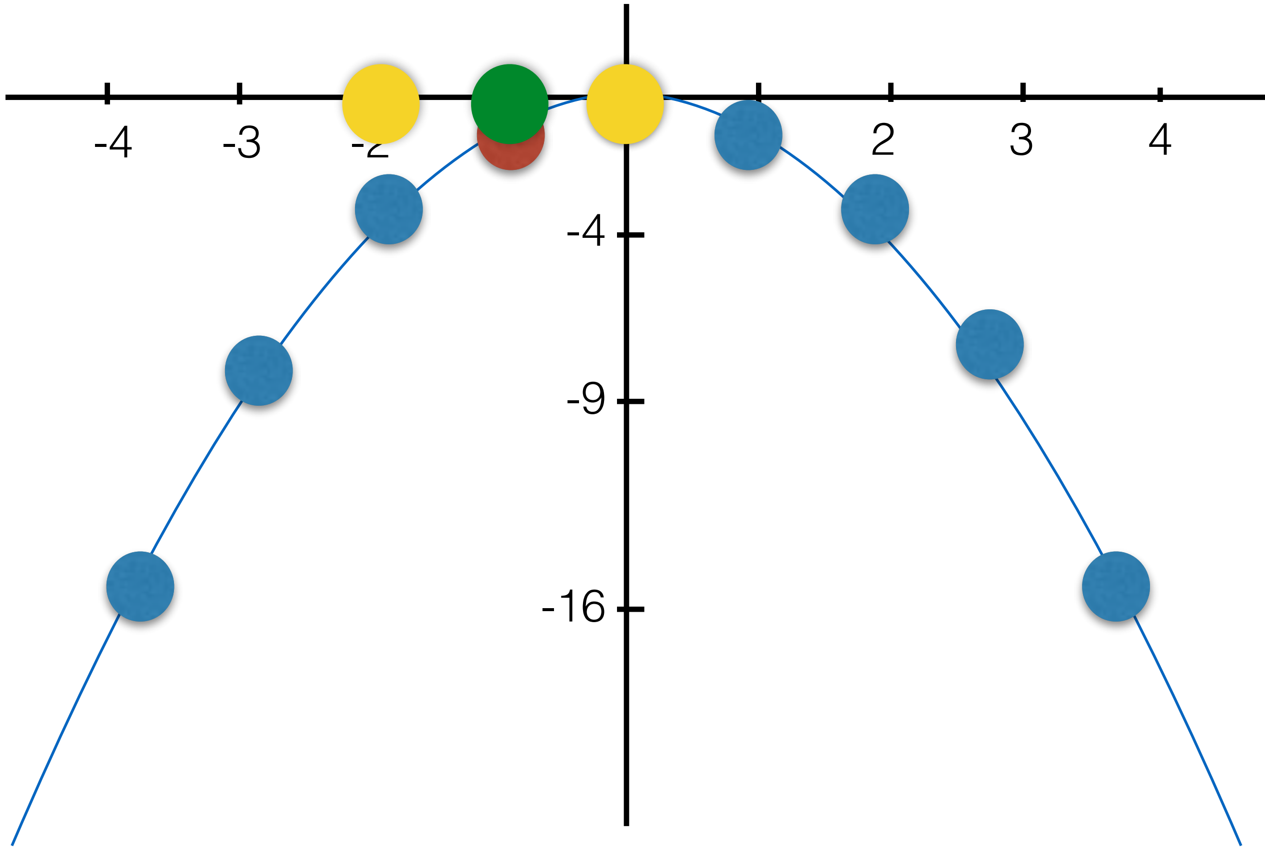
```
1. current_solution = generate initial solution randomly

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)

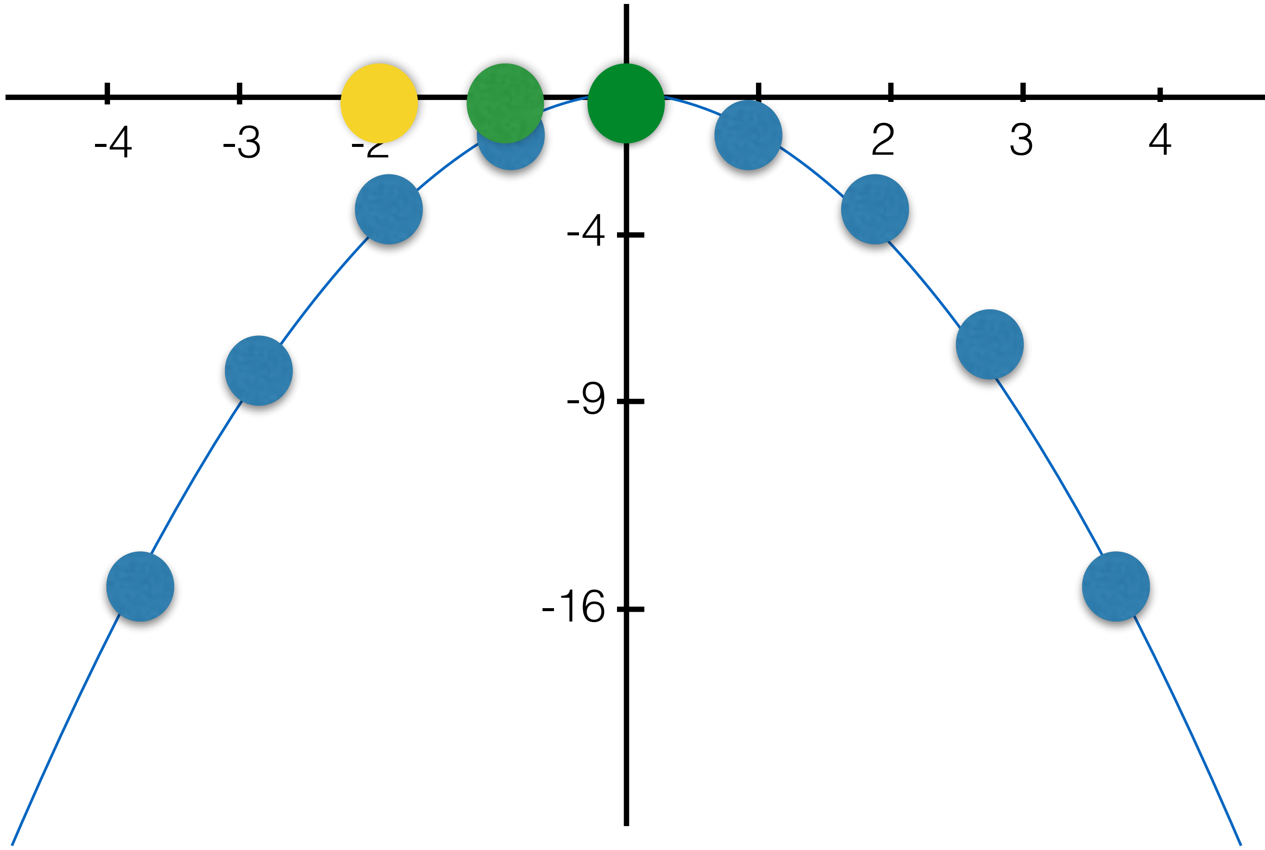
```
1. current_solution = generate initial solution randomly

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)

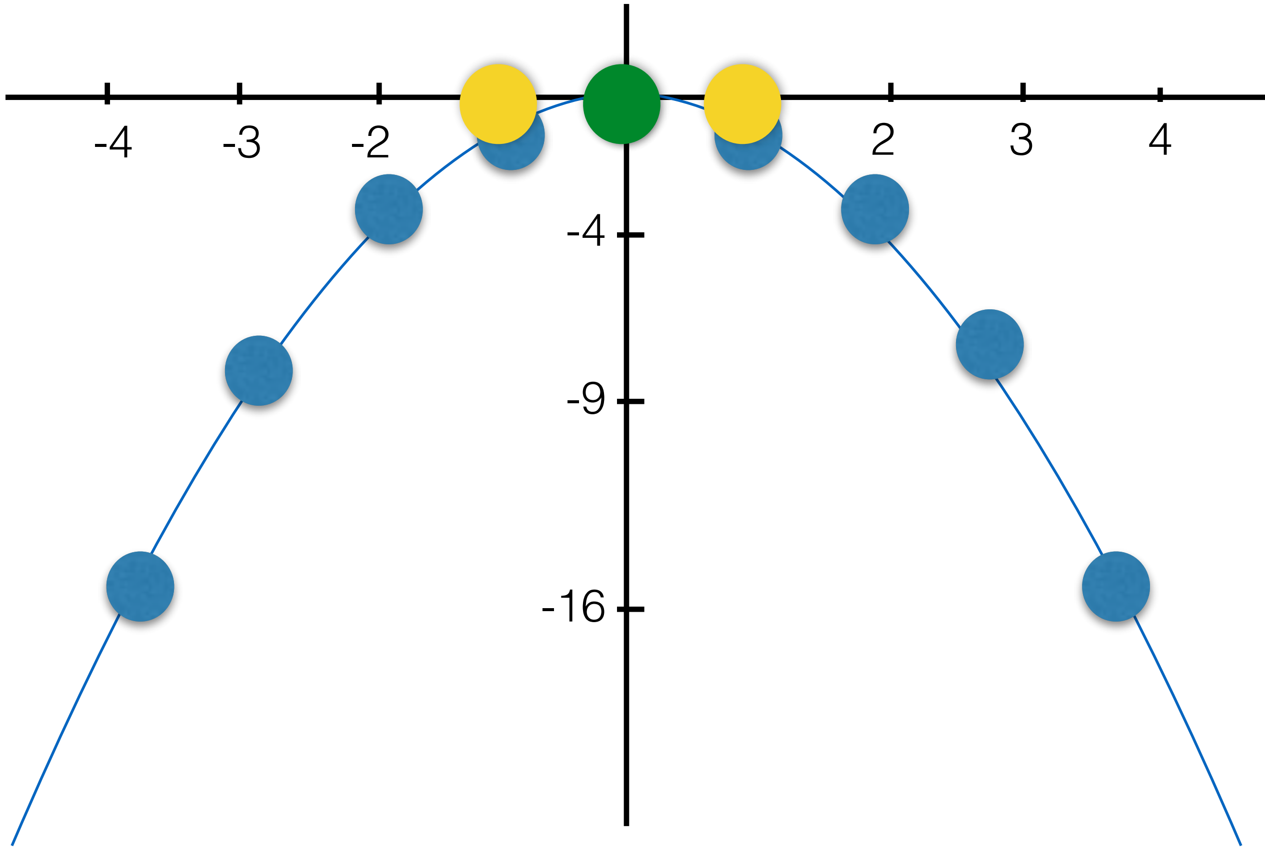
```
1. current_solution = generate initial solution randomly

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
```



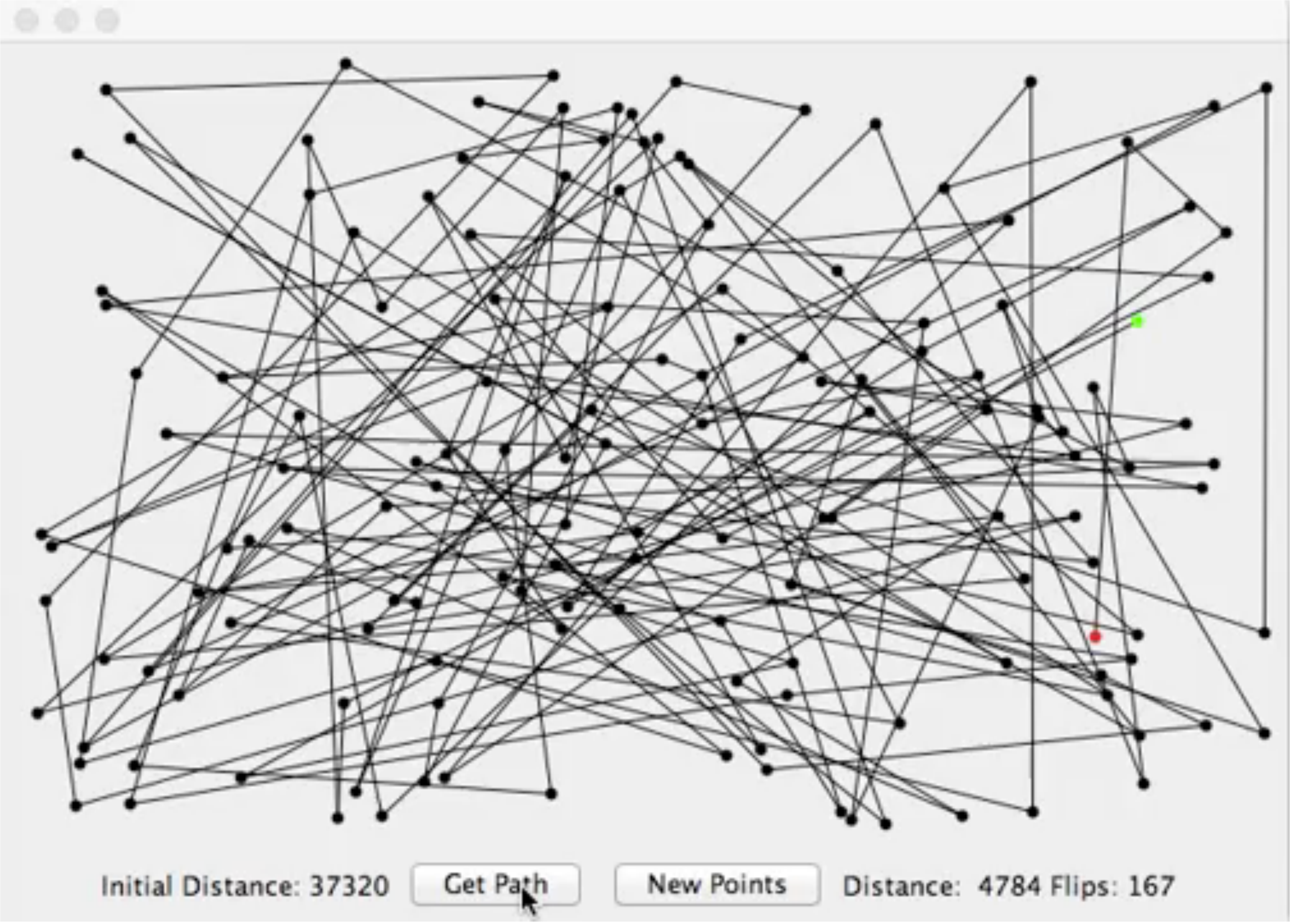
Traveling Salesman Problem Formulation

- Design variables represent a solution.
  - Vector **x** of size *N*, where *N* is the number of cities.
  - **x** represents a sequence of cities to be visited.

- Design variables define the search space of candidate solutions.
  - All possible sequences of cities, where each city appears only once.
  - Neighbourhood: reverse path between two cities in the sequence.

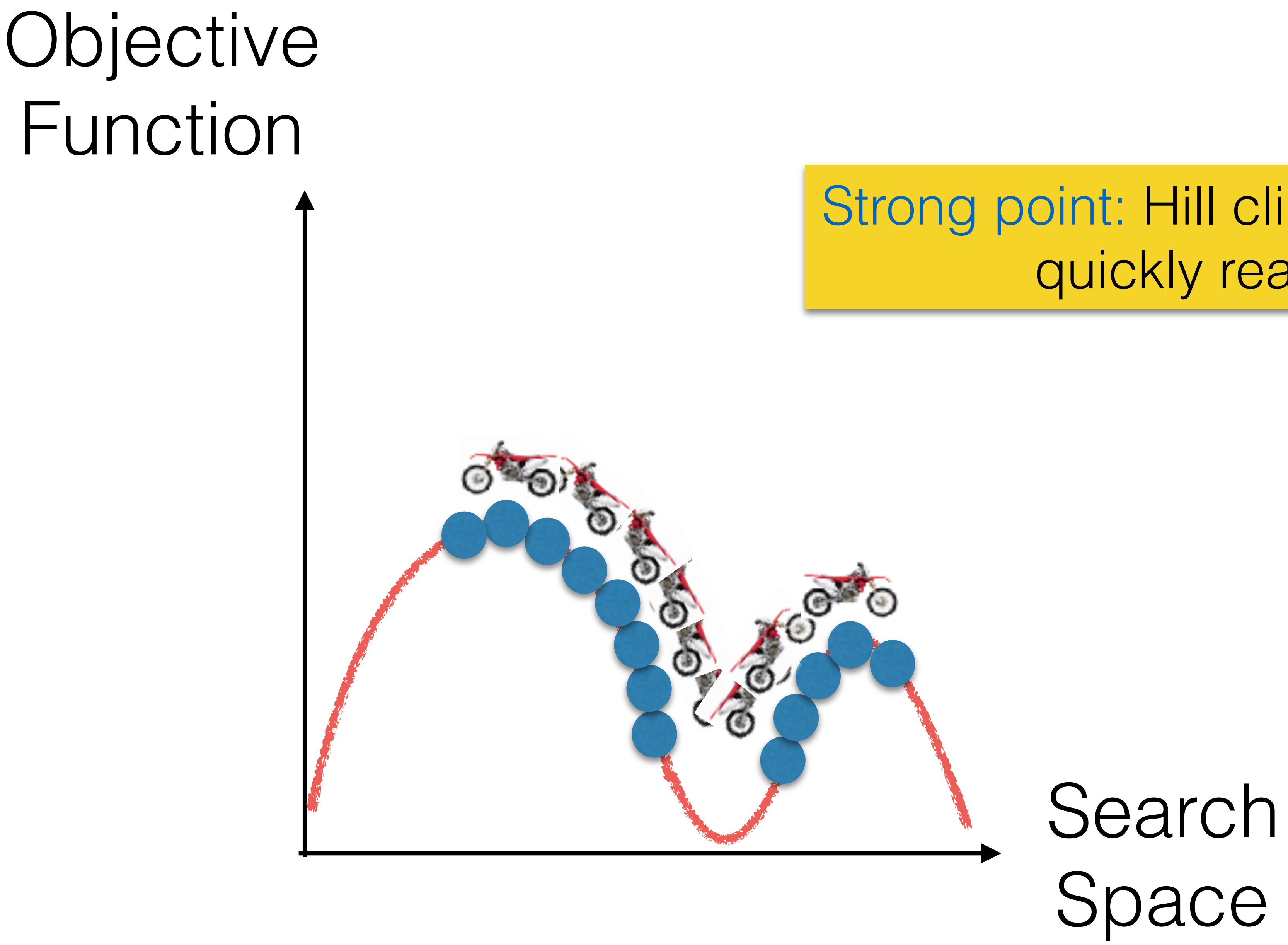
- [Optional] Solutions must satisfy certain constraints.
  - Each city must appear once and only once in **x**.
  - Salesman must return to the city of origin.

- Objective function defines our goal.
  - Total_distance(**x**) =
    sum of distances between consecutive cities in **x** + distance from last city to the origin.
  - To be minimised.



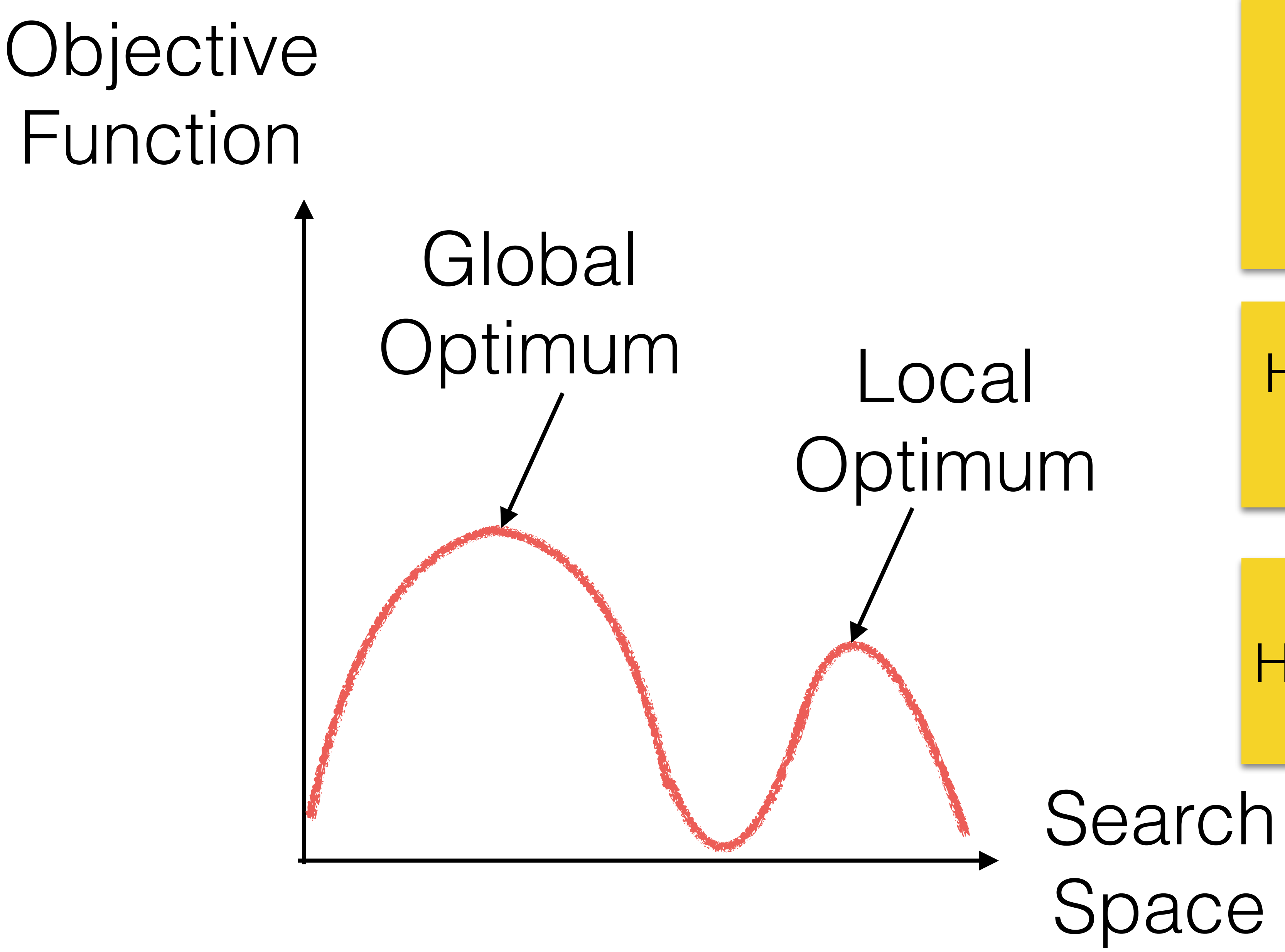
[Video posted by sarahbau: https://youtu.be/3TrnjUKeFg8

General Idea



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

Greedy Local Search

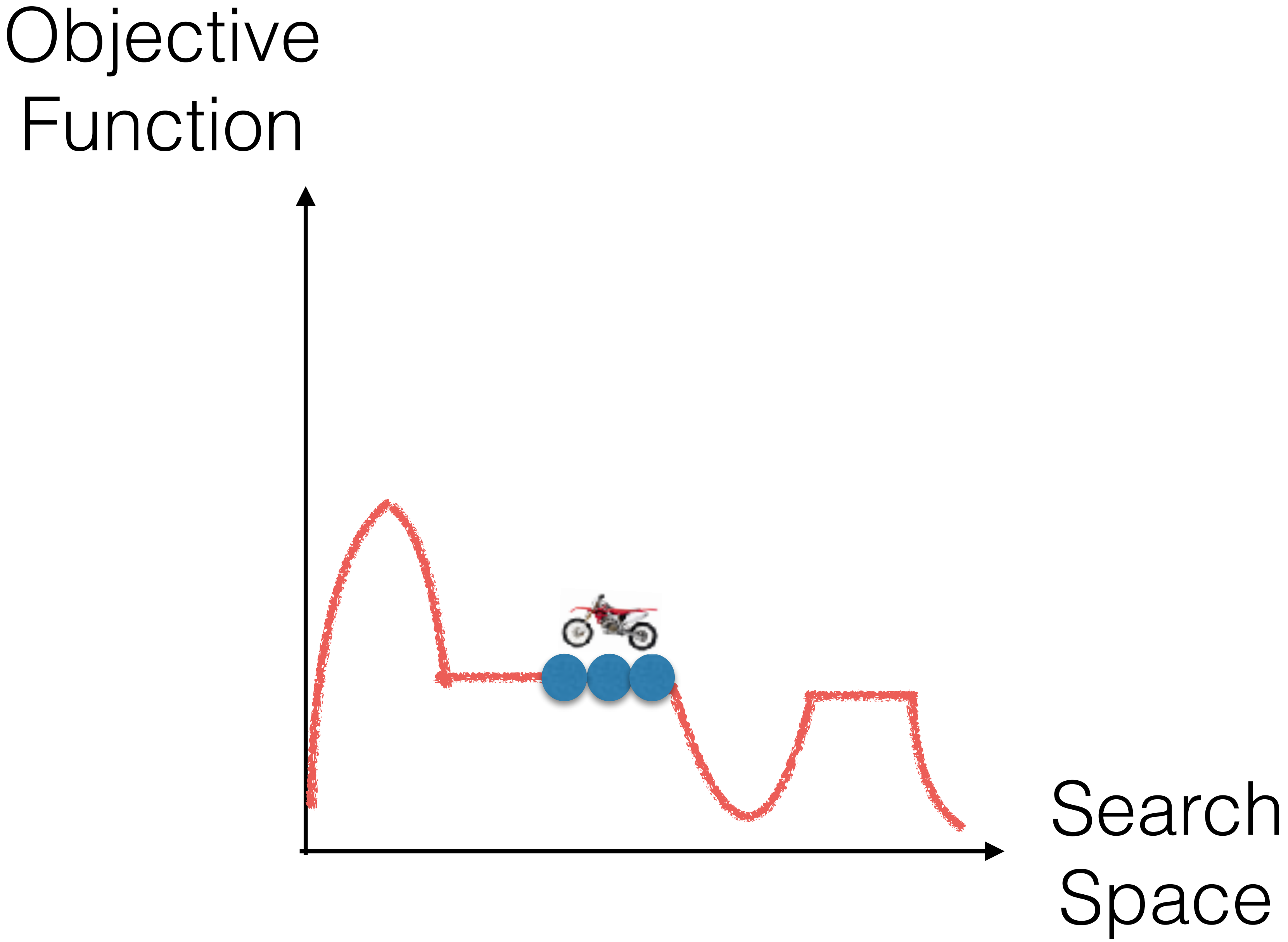


Weakness: Hill-climbing may get trapped in a local optimum.

Hill-climbing is a local search method.

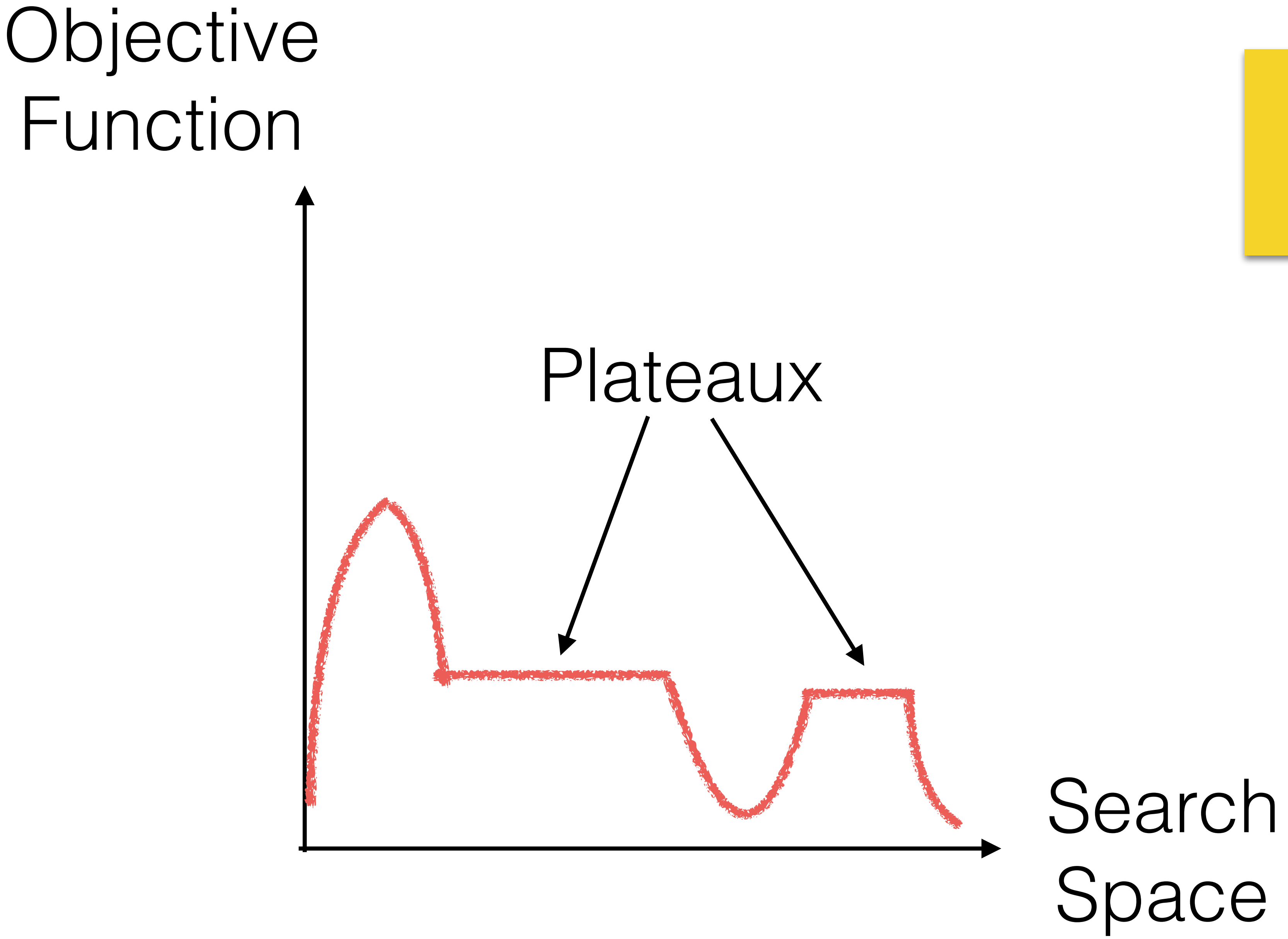
Hill-climbing is greedy.

Greedy Local Search



Greedy Local Search

Weakness: Hill-climbing may get trapped in plateaux.



The success of hill-climbing depends on the shape of the quality function for the problem instance in hands.

Summary

- How to formulate optimisation problems.
- Brute-force search.
- How hill-climbing works.
- Problems of hill-climbing.

Examples of Software Engineering Optimisation Problems

- Hill-climbing has been successfully applied to software module clustering.

- Software Module Clustering:
  - Software is composed of several units, which can be organised into modules.
  - Well modularised software is easier to develop and maintain.
  - As software evolves, modularisation tends to degrade.

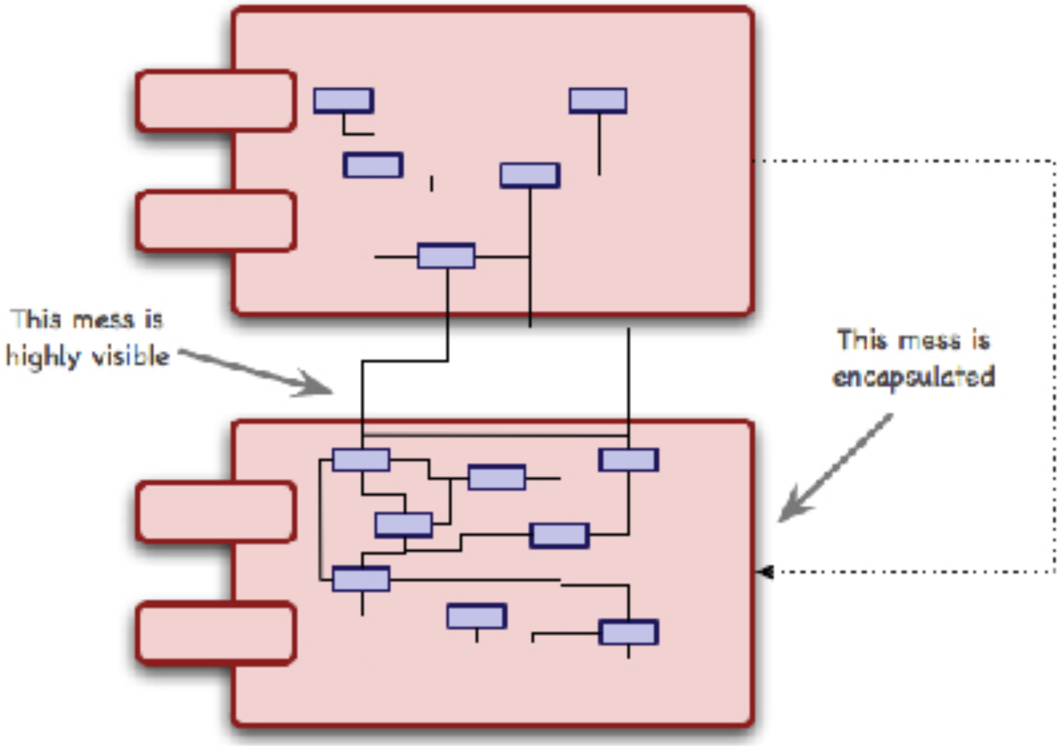


Image from: http://www.kirkk.com/modularity/wp-content/uploads/2009/12/EncapsulatingDesign1.jpg

Problem: find a grouping of units into modules that maximises the quality of modularisation.

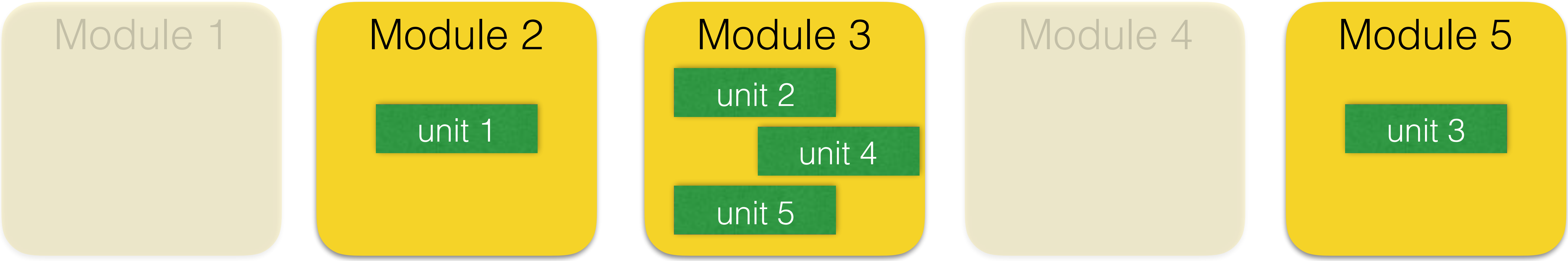
Formulation Optimisation Problems

- Design variables represent a solution.
- Design variables define the search space of candidate solutions.
- Objective function defines our goal.
  - Can be used to evaluate the quality of solutions.
  - Function to be optimised (maximised or minimised).
- [Optional] Solutions must satisfy certain constraints.

Formulating Software Module Clustering as an Optimisation Problem

Design variable: grouping of units into modules.

- Consider that we have *N* units.
- We have at most *N* modules.
- Our variable could be a list of *N* modules, each of which is a set of units.

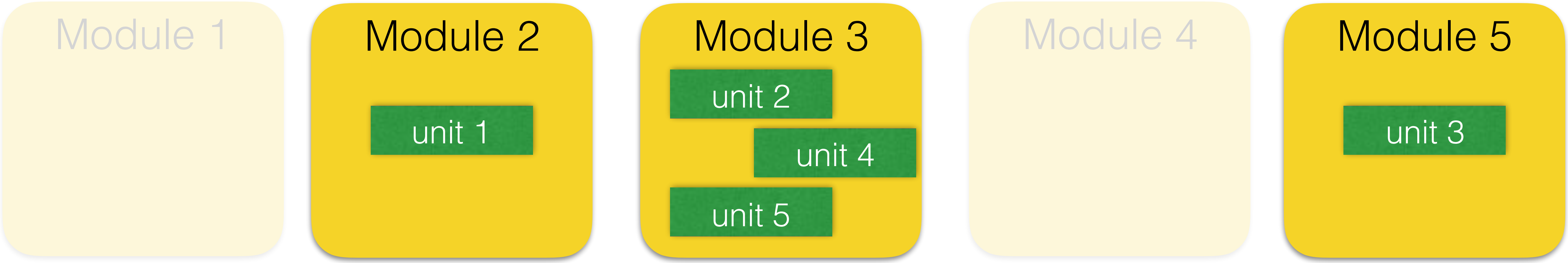


- E.g., if we have *N*=5, a possible grouping is {{},{1},{2,4,5},{},{3}}.

Search space: all possible groupings.

Neighbourhood

- A neighbour would be a solution where a single unit moves from one module to another. E.g.:



Module 1

Module 2
unit 1

Module 3
unit 2
unit 4
unit 5

Module 4

Module 5
unit 3

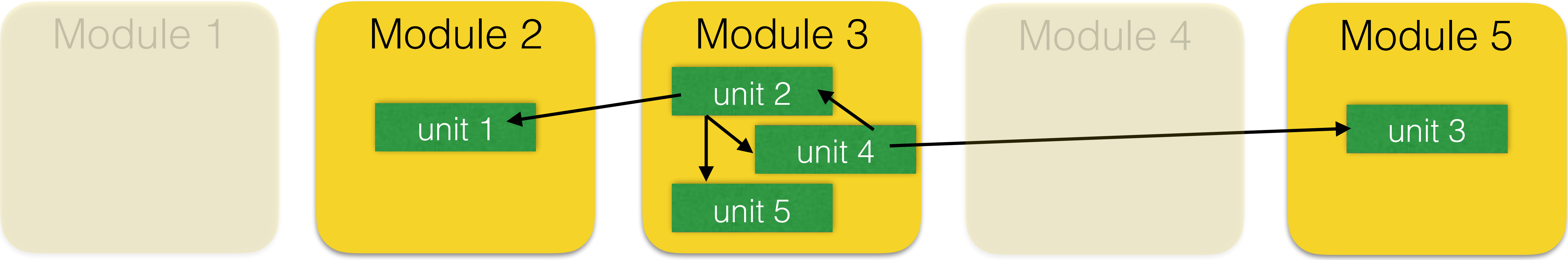
Formulating Software Module Clustering as an Optimisation Problem

Constraints: N/A

Objective function: quality of modularisation (to be maximised).

How to compute quality?

What does good quality mean?



Lots of connections inside a module (high cohesion) and few connections between modules (low coupling).

$$\uparrow \text{Quality of module i} = \frac{\#\text{intra\_edges\_i} \uparrow}{\boxed{\#\text{intra\_edges\_i}} + 1/2 \times \#\text{inter\_edges\_i} \downarrow}$$

(maximise)

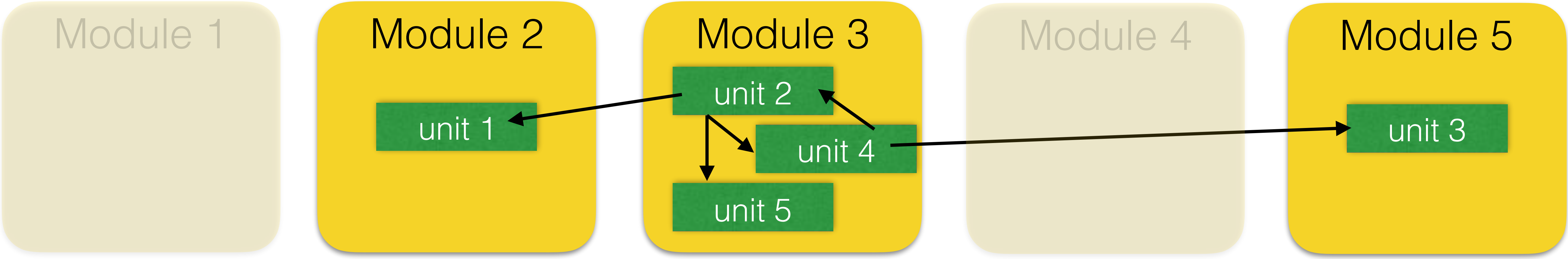
1/2 is used to split the penalty of inter-edges across the two modules connected by that edge

Formulating Software Module Clustering as an Optimisation Problem

Objective function: quality of modularisation (to be maximised).

How to compute quality?

What does good quality mean?



Lots of connections inside a module (high cohesion) and few connections between modules (low coupling).

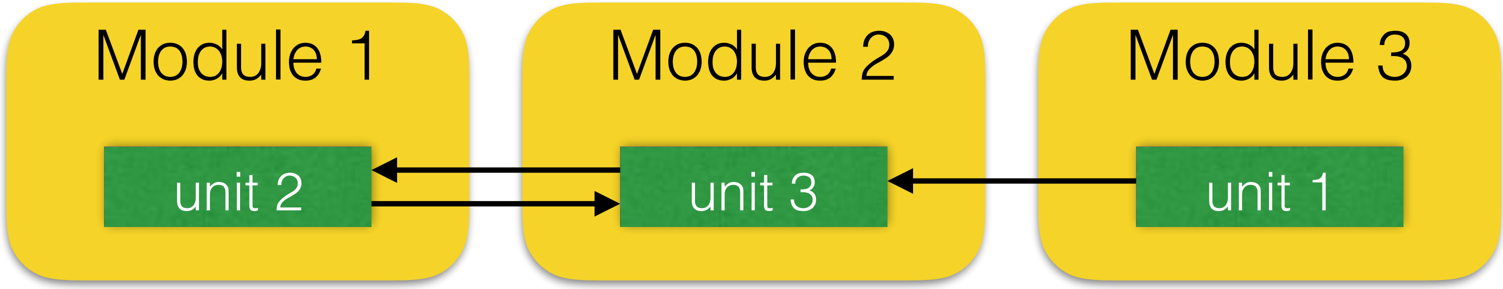
Quality of module i (maximise) $= \dfrac{\#intra\_edges\_i}{\#intra\_edges\_i + 1/2 \times \#inter\_edges\_i}$

Quality of a solution = sum of quality of its non-empty modules

Example of Hill-Climbing for Software Module Clustering

Number of units N = 3

1. current_solution = generate initial solution randomly

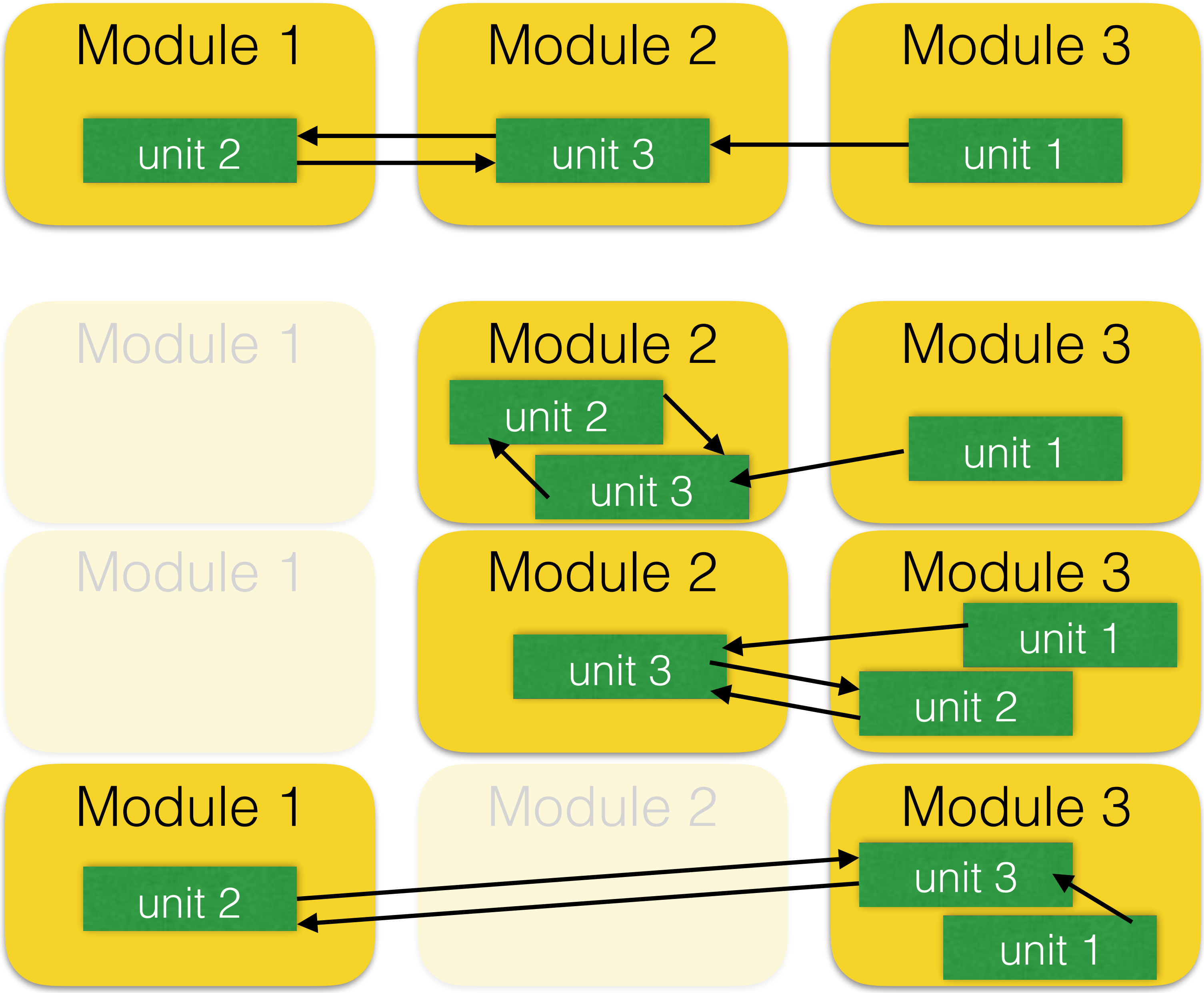


current_solution
{2}{3}{1}

Connections between units can be retrieved automatically from the software code.

Example of Hill-Climbing for Software Module Clustering

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



current_solution
{2}{3}{1}

neighbours
{}{2,3}{1}

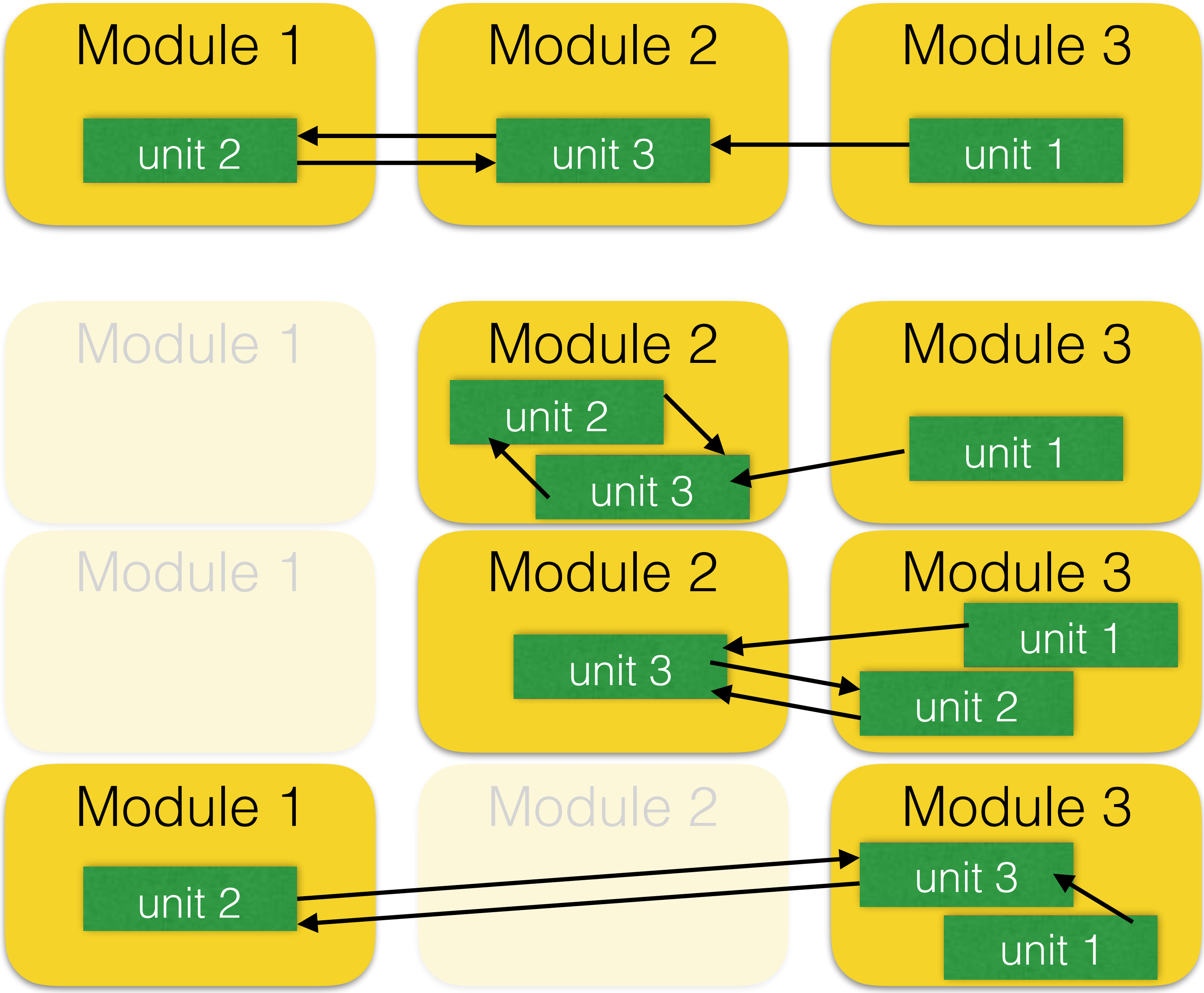
{}{3}{1,2}

{2}{}{1,3}

All solutions with a single unit moving to a different module

Example of Hill-Climbing for Software Module Clustering

2.2 best_neighbour = get highest quality neighbour of current_solution



current_solution
{2}{3}{1} quality = 0

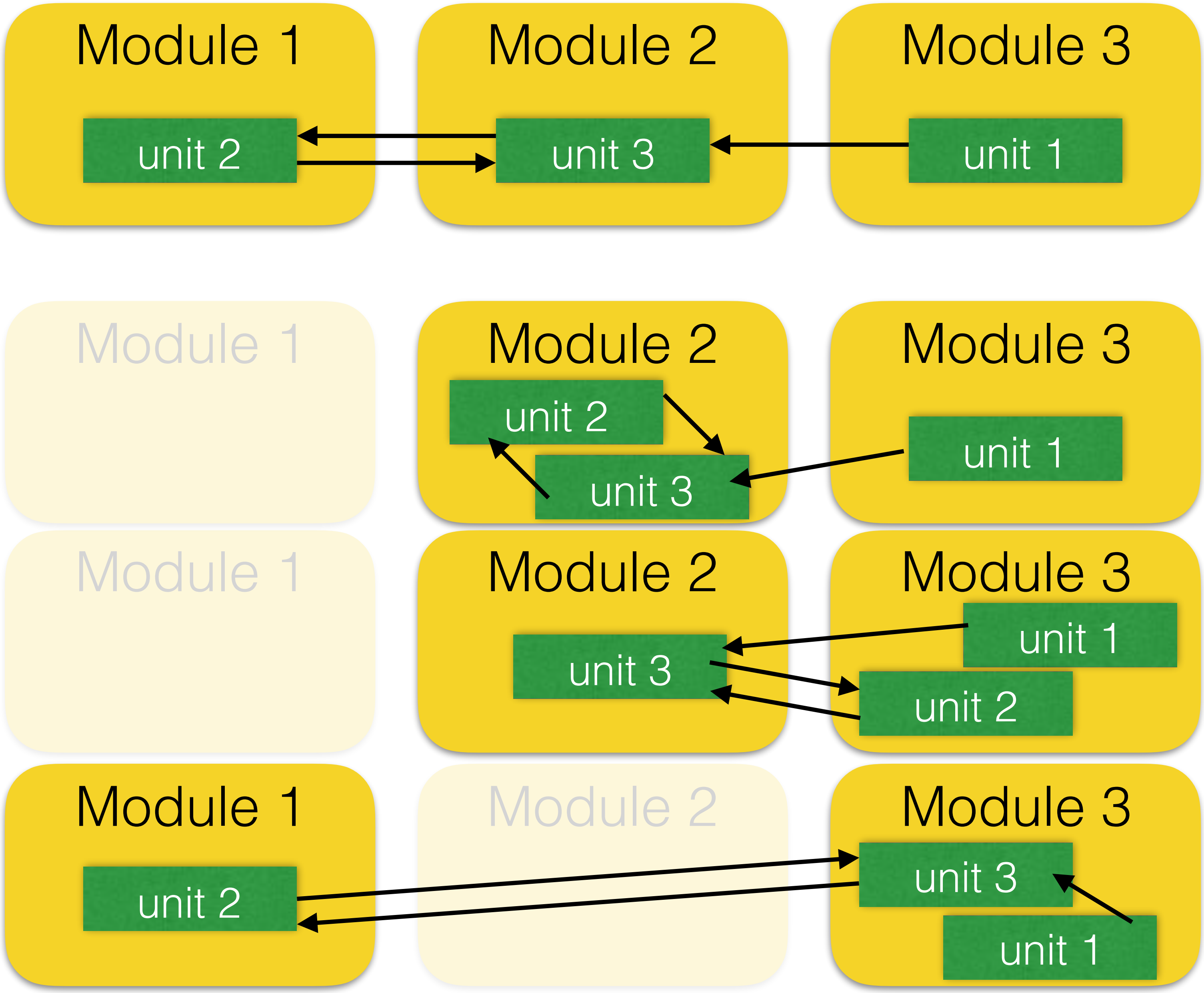
neighbours
{}{2,3}{1} quality = 0.8

{}{3}{1,2} quality = 0

{2}{}{1,3} quality = 0.5

Example of Hill-Climbing for Software Module Clustering

```
2.3 If quality(best_neighbour) <= quality(current_solution)
    2.3.1 Return current_solution
```



current_solution
{2}{3}{1} quality = 0

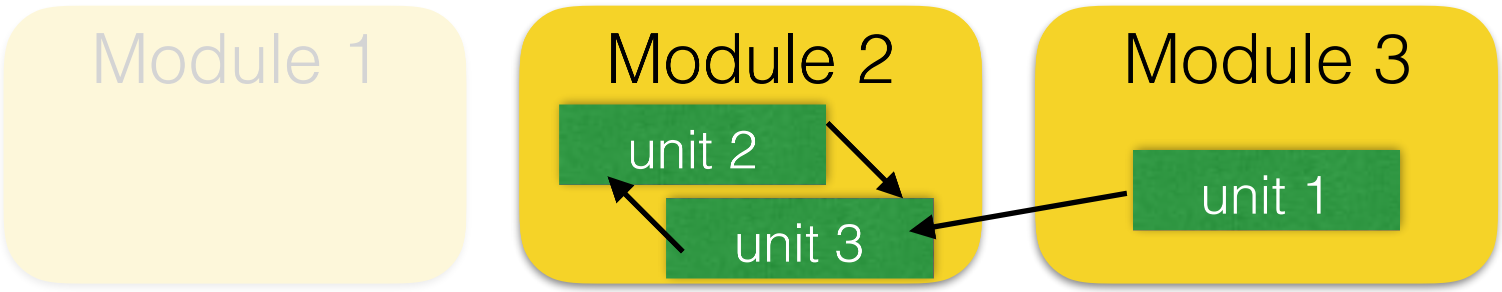
neighbours
{}{2,3}{1} quality = 0.8

{}{3}{1,2} quality = 0

{2}{}{1,3} quality = 0.5

Example of Hill-Climbing for Software Module Clustering

2.4 current_solution = best_neighbour



current_solution

{}{2,3}{1} quality = 0.8

Further Reading

All material available from Reading Lists (http://readinglists.le.ac.uk/lists/D888DC7C-0042-C4A3-5673-2DF8E4DFE225.html)

Stuart J. Russell, Peter Norvig, John F. Canny
Artificial intelligence: a modern approach
Section 4.1: Local Search Algorithms and Optimization Problems - Hill-Climbing Search
Pearson Education
2014

Brian S. Mitchell and Spiros Mancoridis
Using Heuristic Search Techniques to Extract Design Abstractions from Source Code
Sections 3 up to the end of section 3.1.1
Proceedings of the Genetic and Evolutionary Computation Conference
Pages 1375-1382
2002