Lecture 05 – Metaheuristics

Optimisation CT5141

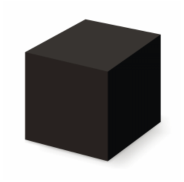
James McDermott

University of Galway

Overview

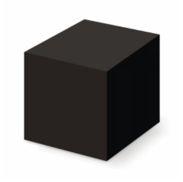
1. **Metaheuristics**
2. Hill climbing
3. Search spaces and search landscapes
4. Smarter hill-climbing

Black-box objective functions


Source

A **black-box** function is a function f which we can **run** (can call $f(x)$), but we can't **read**

Black-box objective functions



Source

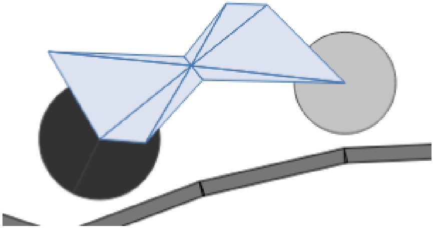
A **black-box** function is a function f which we can **run** (can call $f(x)$), but we can't **read**

OR, we have the code, but it's so complex we can't see any easy relationship between input and output.

No constraints?

Often in such problems we don’t have any constraints, or they are encoded just as penalties in the objective. So in this section we will mostly consider problems with no constraints.

Example: simulation



rednuht.org

Objective: distance travelled in the simulation.

A car is defined by:

- Shape (8 floats, 1 per vertex)
- Wheel size (2 floats, 1 per wheel)
- Wheel position (2 ints, 1 per wheel)
- Wheel density (2 floats, 1 per wheel) darker wheels mean denser wheels
- Chassis density (1 float) darker body means denser chassis

Definitely not a linear objective! In fact it is hard to say anything *a priori* about what DV values are good, or about the gradient.

Metaheuristics

When the objective is a complex, black-box function of the input, we resort to **weak** solver methods, such as **metaheuristics**.

Definition: a metaheuristic is a search method that uses some generic (not problem-specific) randomised strategy to guide search, using only the objective values of the complete solutions observed so far.

(i.e., treating the objective as a black-box)

Examples:

- Hill-climbing
- Simulated annealing
- Genetic algorithms
- Particle swarm

Metaheuristic optimisation = black art

Metaheuristics are **randomised** and there is **no guarantee** they will find the optimum.

Metaheuristic optimisation = black art

Metaheuristics are **randomised** and there is **no guarantee** they will find the optimum.

Thus, they are not true algorithms! (But we often call them algorithms for convenience.)

Metaheuristic optimisation = black art

Metaheuristics are **randomised** and there is **no guarantee** they will find the optimum.

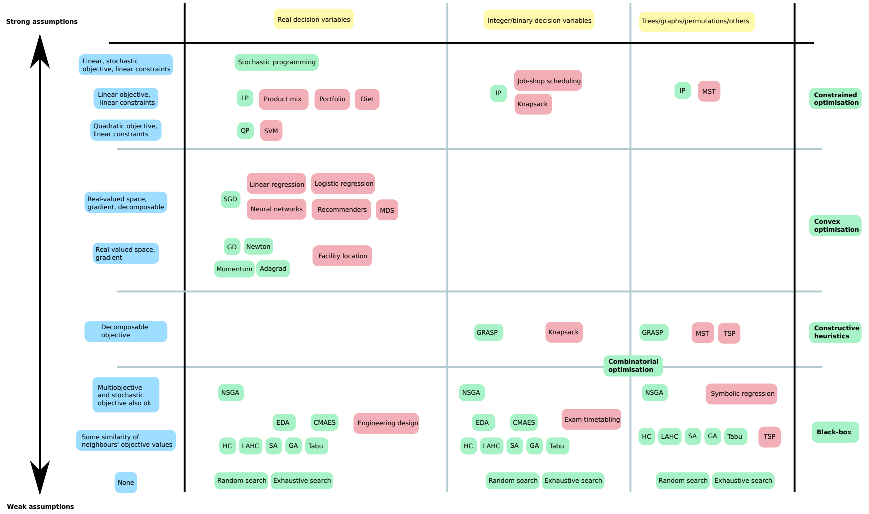
Thus, they are not true algorithms! (But we often call them algorithms for convenience.)

Making them work sometimes relies on unwritten knowledge, rules of thumb, creativity…

Optimisation Landscape
James McDermott
MSc Artificial Intelligence
National University of Ireland, Galway

Legend Assumptions | Search space type | Applications | Techniques

OÉ Gaillimh
NUI Galway

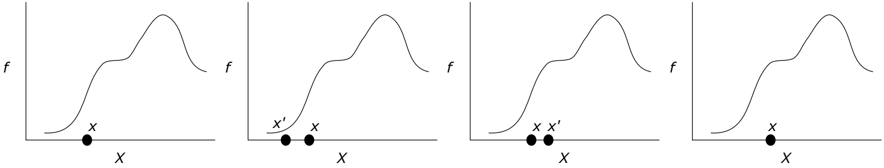


Overview

1 Metaheuristics
2 **Hill climbing**
3 Search spaces and search landscapes
4 Smarter hill-climbing

Hill-climbing, the simplest metaheuristic

Idea: choose a random point x as our **current point**. Then try out another randomly-chosen point x' **near to** x. If $f(x') > f(x)$, then **move** to x', i.e. set $x := x'$. Repeat.



Hill-climbing, the simplest metaheuristic

```python
def hill_climb(f, init, nbr, its):
    """
    f: objective function X -> R
    init: function giving random element x of X
    nbr: function X -> X, giving a neighbour of x
    its: number of iterations
    return: best ever x
    """
    x = init() # make a random point
    for i in range(its):
        xnew = nbr(x) # make new point by changing x
        if f(xnew) > f(x): # if it's better
            x = xnew # step to the new point
    return x
```

Near?

- Our idea is to try randomly-chosen points **near** the current point.
- What does “near” mean?
- In the case of bitstrings, “near” means “flip one bit”:

  E.g. `00110110 -> 01110110`
- According to **Hamming distance**, the distance between these two bitstrings is 1 (because one bit differs).
- Or we might just say they are **neighbours**.

Hill-climbing operators

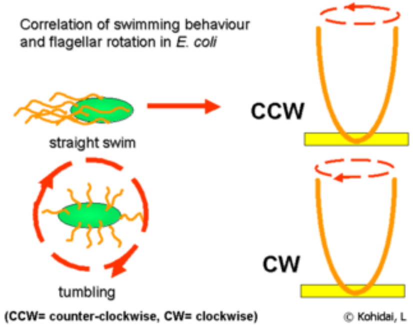
Metaheuristics need two main **operators**:

```python
def bitstring_init(n):
    # uniform sampling from X
    return [random.randrange(2) for i in range(n)]

def bitstring_nbr(x):
    # make a "blind", random change
    x = x.copy() # don't change x itself
    i = random.randrange(len(x))
    x[i] = 1 - x[i]
    return x
```

Chemotaxis

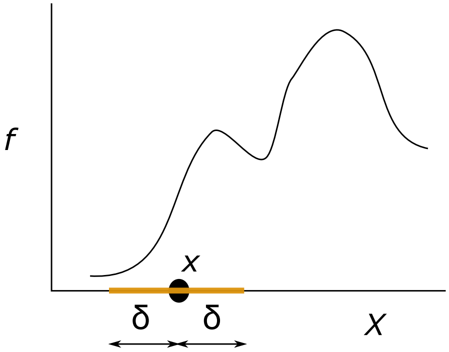
A nice metaphor: bacteria are able to use hill-climbing! E.g. to find food or move away from poisons. They rotate randomly, then swim straight. If they sense disimprovement (lower sugar concentration) they stop swimming straight sooner than if they sense improvement.


Correlation of swimming behaviour and flagellar rotation in *E. coli*
CCW
straight swim
CW
tumbling
(CCW= counter-clockwise, CW= clockwise)
© Kohidai, L

Kohidai, Laszlo (Wikipedia)

Neighbours in real-valued search spaces

- In $\mathbb{R}^1$: a point x is just a number,
- So a neighbour x' of x could be defined as a number such that $|x - x'| < \delta$ for some small δ.
- Or define $x' = x + N(0, \delta)$, where N is a normal distribution with mean 0 and standard deviation δ



Neighbours in real-valued search spaces

```python
def real_init(n):
    return [random.random() for i in range(n)]

def real_nbr(x):
    x = x.copy()
    i = random.randrange(n)
    # add a small constant in range [-delta, delta]
    delta = 0.3
    x[i] += random.random() * 2 * delta - delta
    return x
```

A small detail: DV bounds

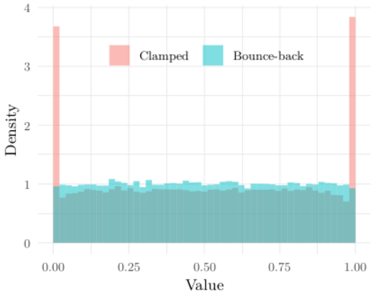
Suppose we have **box constraints**: lower and upper bounds on DVs. What should we do if our `nbr` function gives a value that violates a bound? E.g. suppose UB=1.0 and `nbr` gives 1.1. Options:

- **Clamp** values at the bounds: output = 1.0
- **Bounce-back**, i.e. reflect the value: output = 0.9.

A small detail: DV bounds

Suppose we have **box constraints**: lower and upper bounds on DVs. What should we do if our `nbr` function gives a value that violates a bound? E.g. suppose UB=1.0 and `nbr` gives 1.1. Options:

- **Clamp** values at the bounds: output = 1.0
- **Bounce-back**, i.e. reflect the value: output = 0.9.



Nordmoen et al. (2020) show that bounce-back is better

Complex constraints, penalties and repair

- Sometimes constraints depend on multiple variables at once, e.g. $3x_0 + 2x_1 < 10$
- It might be difficult to design operators which respect that
- Or the operators we design might introduce unintended bias
- Might be easier to apply an **objective penalty** to individuals which don't respect a constraint,
  e.g. $f' := f + \max(0, 3x_0 + 2x_1 - 10)$
- Another is to **repair** it – make some simple change that makes it into a valid genotype.

Exploration and exploitation

Exploration:

- The search algorithm bravely tries out new areas of the search space
- E.g. **random search**

Exploitation:

- The search algorithm concentrates on areas **near** known-good areas
- E.g. **hill-climbing**

Exploration and exploitation

Exploration:

- The search algorithm bravely tries out new areas of the search space
- E.g. **random search**

Exploitation:

- The search algorithm concentrates on areas **near** known-good areas
- E.g. **hill-climbing**

More sophisticated algorithms are in the spectrum between the two.

The great dilemma of black-box optimisation

Exploration versus exploitation in $\mathbb{R}^n$

- As we just saw, in $\mathbb{R}^n$, we can control the **step-size**, δ
- Small $\delta \rightarrow$ small steps $\rightarrow$ more exploitation
- Large $\delta \rightarrow$ large steps $\rightarrow$ more exploration

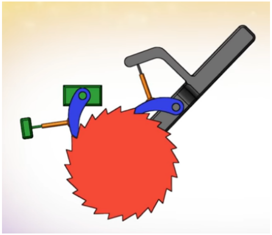
Why does hill-climbing work?

The `nbr` function is **blind** or **undirected**. We do **not** try to design a neighbour operator that gives improvements.

Why does hill-climbing work?

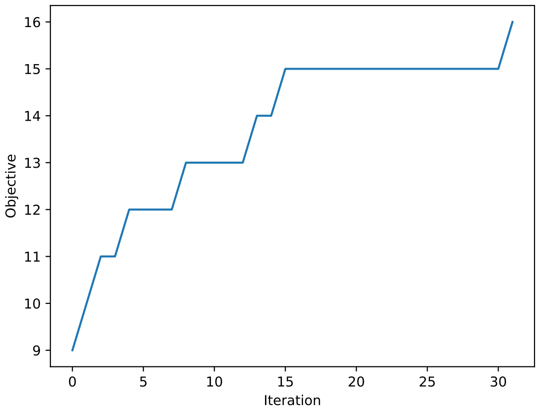
The `nbr` function is **blind** or **undirected**. We do **not** try to design a neighbour operator that gives improvements.

The power of hill-climbing is from the **selection** (the `if`-statement): we only accept moves which **improve**.



The ratchet, from https://youtu.be/EpVPG2fZrHE?t=8

Why does hill-climbing work?



Thanks to the ratchet, the objective value of the best individual improves **monotonically** (never disimproves).

When is hill-climbing better than random search?

Hill-climbing makes weak assumptions:

- Suppose whenever x_1 and x_2 are **similar** in some way, their objective values are also similar
- If $|x_1 - x_2|$ is small, then $|f(x_1) - f(x_2)|$ is small (reminiscent of defn of **continuity** in calculus)
    - Then if we have found a **good** x, it makes sense to try out some (randomly-chosen) **neighbours**, not points randomly sampled from the entire space.
    - (E.g. Wallace and Aleti, *The Neighbours' Similar Fitness Property for Local Search*)
- Our `nbr` function can always find an improvement (eventually) and so make progress.

Quiz

Hill-climbing will maximise a function f. If I want to minimise instead:

- I need a different algorithm
- I can use hill-climbing to maximise $-f$
- I need to calculate the gradient of f
- I'm out of luck because minimisation is impossible

Quiz

Hill-climbing will maximise a function f. If I want to minimise instead:

- I need a different algorithm
- I can use hill-climbing to maximise $-f$
- I need to calculate the gradient of f
- I'm out of luck because minimisation is impossible

Answer: I can use hill-climbing to maximise $-f$

Quiz

Random search is exploitative: True or False?

Quiz

Random search is exploitative: True or False?

Answer: no, in fact RS is the most explorative algorithm.

Overview

1. Metaheuristics
2. Hill climbing
3. **Search spaces and search landscapes**
4. Smarter hill-climbing

When does hill-climbing fail?

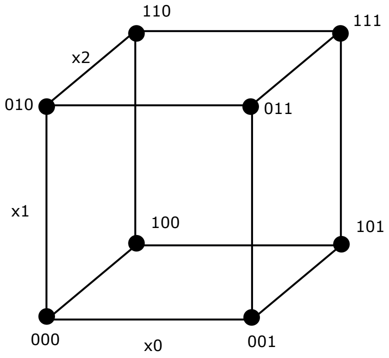
To answer this question, we need to understand the concepts of **search landscapes** and **local optima**. But before we can define **search landscape** we have to recall **search spaces**.

Search spaces: Cartesian spaces

- This is the most common type, as in LP and IP
- We have n **decision variables** $x_0, \ldots, x_{n-1}$
- Usually all of same type, e.g. binary, integer, categorical, or real (possibly with bounds)
- Then X is the **Cartesian product** of the decision variables.
- Items in the search space are then **vectors**.

Hamming cube

$X = \mathbb{H}^n$ (Hamming cube in n dimensions, i.e. space of bitstrings of length n)

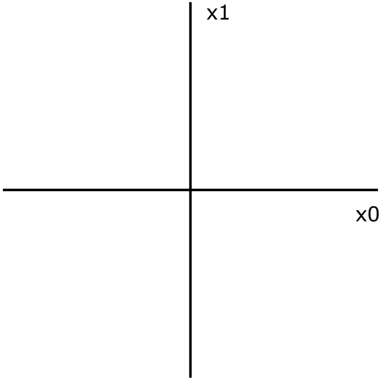


Examples:

- Binary guessing game (Week 01)
- Binary Unit Commitment: a single power plant is on or off at n intervals
- Feature Selection: given n features we choose a subset for machine learning.

Real vector spaces

$X = \mathbb{R}^n$ (Cartesian space in n dimensions, i.e. space of real vectors of n components)

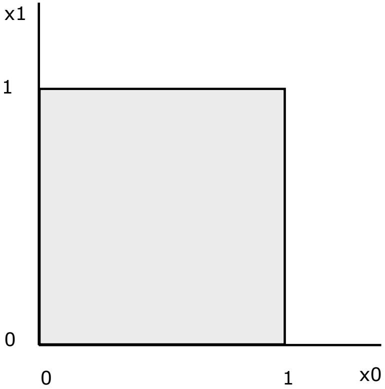


Examples:

- Logistic Regression (the parameters w_i are the optimisation DVs)
- Portfolio allocation (the amount to invest in each stock is a real vector)

Unit hypercube

$X = [0, 1]^n$ (Unit hypercube in n dimensions, a subset of $\mathbb{R}^n$)



Example:

- Portfolio allocation, but all real parameters are **bounded to be in some range**, e.g. [0, 1].

Search spaces: other cases

Above the search space was a Cartesian product of DVs.

But the search space could be **any set**, e.g. a particular set of trees, or permutations, or graphs, or something else. We'll see these later in the module.

What is a landscape?

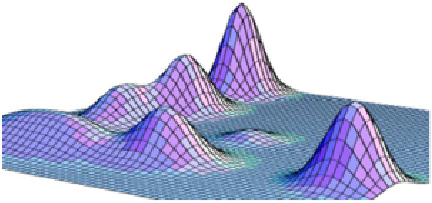
A search **landscape** is a search space X, together with a neighbourhood function `nbr`, and an objective function f.

Landscape is a triple: (X, `nbr`, `f`).

Metaphor

Imagine search as trying to walk around in the mountains, when you can't see far, and only know the height of points you have already visited.

- The search space is the north-south and east-west axes.
- The neighbour function is your footstep.
- The objective function is altitude.

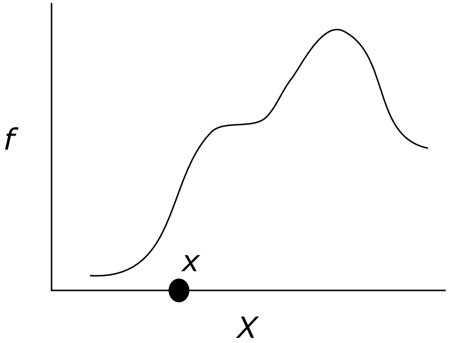


From here

The **landscape** metaphor is due to Sewall Wright.

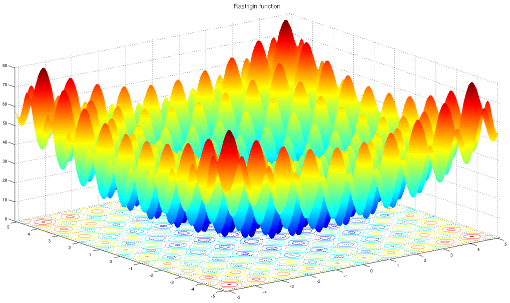
A landscape on 1-D Euclidean space

- 1-D Euclidean space $\mathbb{R}^1$
- Neighbour function just adds a small random number (positive or negative)
- Objective function as shown.



A landscape on 2-D Euclidean space

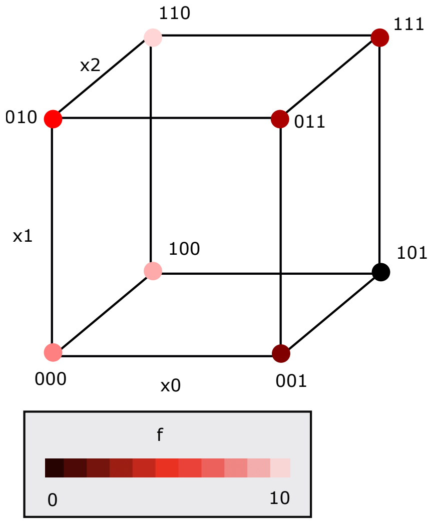
- 2-D Euclidean space $\mathbb{R}^2$
- Neighbour function adds a random number to a randomly-chosen coordinate
- E.g. Rastrigin function, a well-known test function.



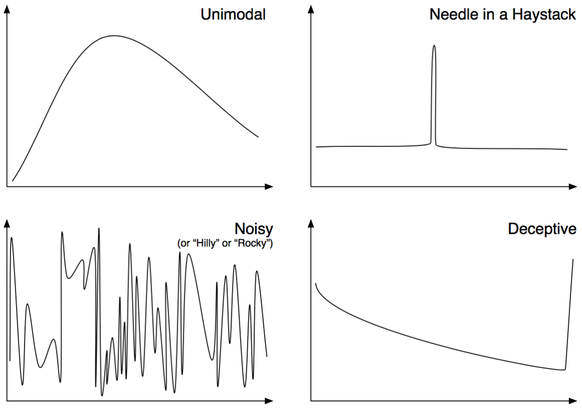
Rastrigin function, image by Diegotorquemada

A landscape on the Hamming cube

- The Hamming cube X = `{000, 001, 010, 011, 100, 101, 110, 111}`
- Neighbour function is "flip one bit"
- Objective function as shown.

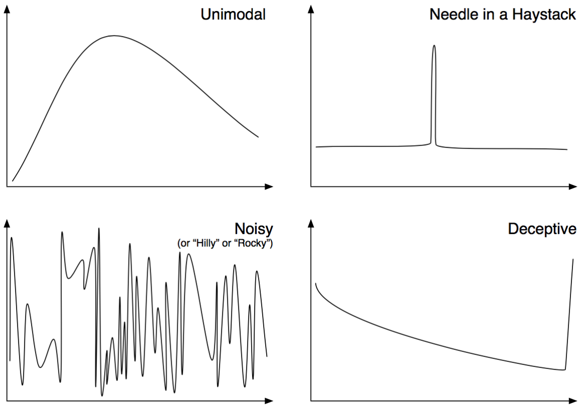


Different types of landscapes



Luke, *Essentials*

Different types of landscapes



Luke, *Essentials*

Reminder: in reality we **can't see** a plot of the landscape!

Landscape properties

- Unimodal (one peak) versus multimodal (many peaks)
- Smooth (continuous, or at least neighbours usually have similar values) versus rugged
- Informative versus deceptive (“gradient” leads towards the global optimum, or away)

Quiz

Consider an LP **maximisation** problem with 2 real DVs. What is the "mountain landscape" like? Is the objective function unimodal? Is it rugged? Is it deceptive?

Quiz

Consider an LP **maximisation** problem with 2 real DVs. What is the "mountain landscape" like? Is the objective function unimodal? Is it rugged? Is it deceptive?

The landscape would look like a subset of an inclined plane, e.g. think of walking up **one face** of a pyramid. It is unimodal, not rugged, not deceptive.

When does hill-climbing fail?

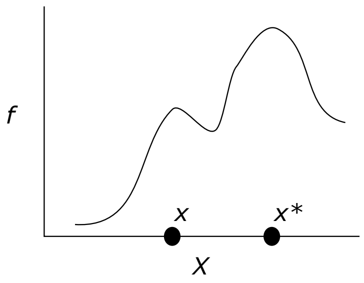
Hill-climbing fails on problems which are multimodal, rugged, and/or deceptive.

When does hill-climbing fail?

Hill-climbing fails on problems which are multimodal, rugged, and/or deceptive.

All of these involve **local optima** in some way.

Global and local optima



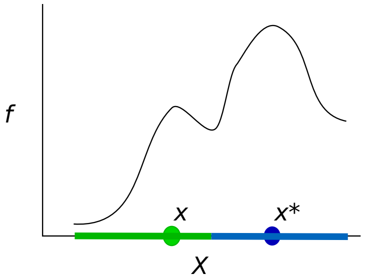
A **global optimum** is a point x^* such that $f(x^*) \geq f(y)\ \forall\ y \in X$. There may be multiple global optima (all equally good).

A **local optimum** is a point x which is **not a global optimum** such that $f(x) \geq f(y)\ \forall$ neighbours y of x.

Global and local optima

Global and local optima on a trampoline

Basins of attraction



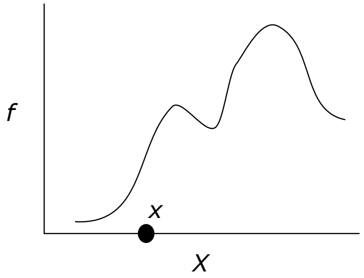
- Each optimum has a **basin of attraction**.
- A basin of attraction B_i is a subset of the search space corresponding to a particular optimum x_i
- If the current point $x \in B_i$, then the hill-climb will end at x_i.

Escaping local optima

Suppose we reach a local optimum. What would we need to do to **escape**...

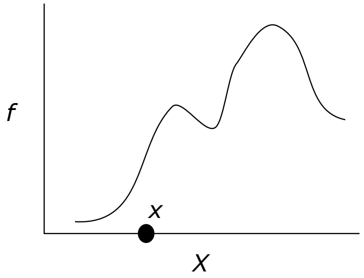
1. ... in a **bitstring** problem?
2. ... in a **real-valued** problem?

How do local optima arise?



- If a DV has a non-linear effect, e.g. quadratic or cubic, or something weirder (e.g.: pretend the above image is a high-order polynomial in a single DV *x*).

How do local optima arise?



- If a DV has a non-linear effect, e.g. quadratic or cubic, or something weirder (e.g.: pretend the above image is a high-order polynomial in a single DV *x*).
- If there is a dependency in the objective function between two or more DVs (epistasis).

Dependency between variables

Why don't we just try all possible values for `x0` and choose the best; then all values for `x1` and choose the best; and so on?

Example: Cookies

- "finding the most delicious chocolate chip cookie recipe from a parameterized space of recipes"
- "Parameters included baking soda, brown sugar, white sugar, butter, vanilla, egg, flour, chocolate, chip type, salt, cayenne, orange extract, baking time, and baking temperature."
- From Golovin et al., *Google Vizier*, KDD '17

Dependency between variables

This example illustrates the issue of **dependency** between variables. We would not say “I’m going to try varying the amount of butter (say), and then I’ll know the right amount of butter. Then I’ll proceed to optimise the sugar in the same way. then the baking temperature, and the number of eggs, etc.”. **This doesn’t work**, because the optimum amount of butter depends on the amount of eggs, etc. We might reach a local optimum.

When there is any dependency between variables, we have to optimise them all together.

Quiz

The objective function in cookie optimisation seems subjective. How could we measure it in a (somewhat) objective way?

Quiz

The objective function in cookie optimisation seems subjective. How could we measure it in a (somewhat) objective way?

For each possible recipe we could bake a large batch and run a survey.

Research topics

An interesting area of research: what makes one problem harder than another? We don't have a formula for it. But we do have some good clues.

- Search space size/dimensionality
- Epistasis (dependency between DVs)
- Landscape ruggedness (many local optima)
- FDC (correlation between objective values and distance from optimum)
- Locality (correlation between objective values of neighbours)

(You don't need to remember these definitions.)

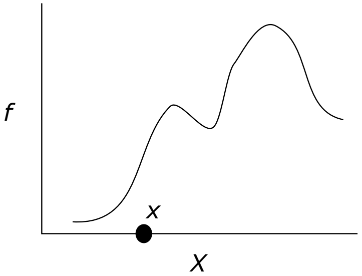
Overview

1 Metaheuristics
2 Hill climbing
3 Search spaces and search landscapes
4 **Smarter hill-climbing**

Recall: Why does hill-climbing fail?

Answer: it gets stuck at local optima.

How to escape local optima



1. Allow **larger jumps** in the `nbr` function
2. **Restart** (**iterated hill-climbing**)
3. Allow **disimproving moves** (**simulated annealing** and **late-acceptance hill-climbing**)
4. Use **multiple search points** with **information transfer** between them (**genetic algorithm**).

Larger jumps in the `nbr` function

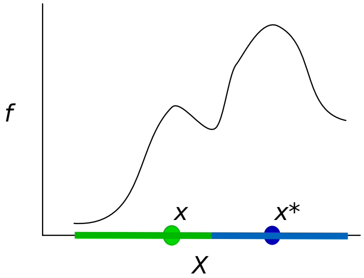
We already saw this when hill-climbing in $\mathbb{R}^n$: we can increase δ to make larger jumps and sometimes escape local optima.

Larger jumps in the `nbr` function

We already saw this when hill-climbing in $\mathbb{R}^n$: we can increase δ to make larger jumps and sometimes escape local optima.

How could we do this in a bitstring problem?

Hill-climbing with restarts



- Idea: start again from random x and hope $x \in B_i$ such that $x^* \in B_i$ also.
- Implementation: just put hill-climbing inside a loop!

Disimproving moves

- Simulated annealing
- Late-acceptance hill-climbing

Simulated annealing

- Simulated annealing is a variant of hill-climbing where we sometimes **accept disimproving moves**, i.e. $x := x'$ even though x' is worse!
- The name **simulated annealing** refers to the **annealing** process used in metallurgy, where cooling a metal slowly can help it reach a better configuration at the atomic level, hence the resulting object is stronger
- Simulated annealing was introduced by Kirkpatrick et al. (1983).

SA: accepting disimprovement

- If $f(x')$ is better than $f(x)$, we always accept x'

SA: accepting disimprovement

- If $f(x')$ is better than $f(x)$, we always accept x'
- A disimproving move is accepted with probability (assume we are **minimising** f):

$$p = \exp\left(\frac{f(x) - f(x')}{T}\right)$$

SA: accepting disimprovement

- If $f(x')$ is better than $f(x)$, we always accept x'
- A disimproving move is accepted with probability (assume we are **minimising** f):

$$p = \exp\left(\frac{f(x) - f(x')}{T}\right)$$

- The probability of accepting disimprovement is small when x' is much worse, but a bit larger when x' is nearly as good as x

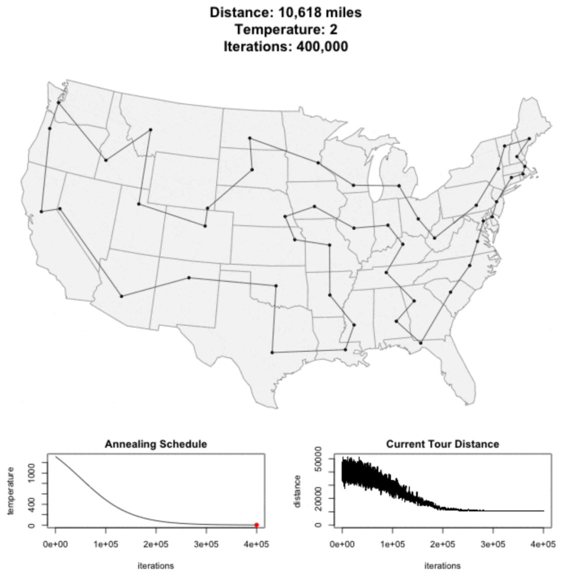
SA: accepting disimprovement

- If $f(x')$ is better than $f(x)$, we always accept x'
- A disimproving move is accepted with probability (assume we are **minimising** f):

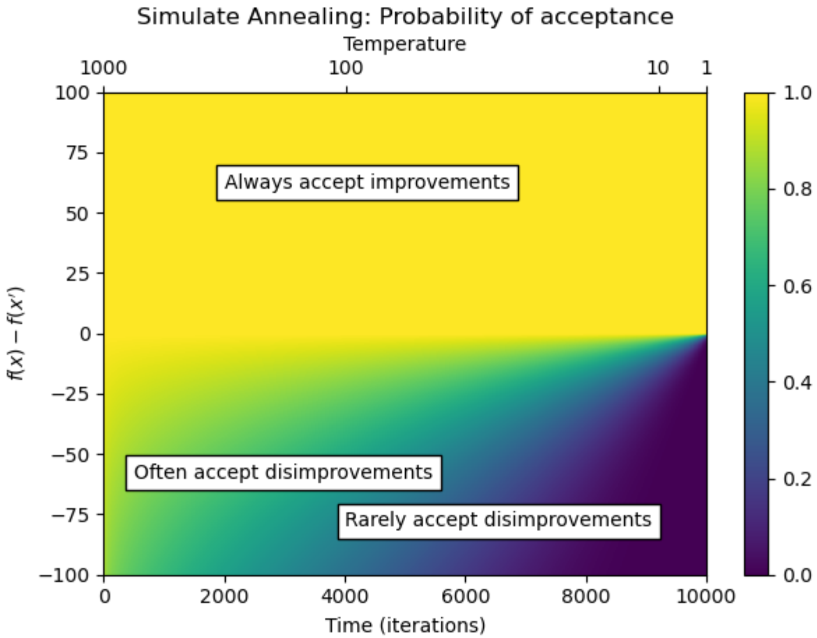
$$p = \exp\left(\frac{f(x) - f(x')}{T}\right)$$

- The probability of accepting disimprovement is small when x' is much worse, but a bit larger when x' is nearly as good as x
- T is **temperature**, which decreases during the run
- The probability of accepting disimprovement is large at the start, but small later.

SA: accepting disimprovement



Temperature decreases ("anneals") over time, as shown here.



Simulate Annealing: Probability of acceptance
Temperature
1000
100
10
1
100
75
50
25
0
−25
−50
−75
−100
$f(x) - f(x')$
Always accept improvements
Often accept disimprovements
Rarely accept disimprovements
0
2000
4000
6000
8000
10000
Time (iterations)
1.0
0.8
0.6
0.4
0.2
0.0

```python
def anneal(f, nbr, init, its):
    # assume we are minimising
    x = init() # initial random solution
    fx = f(x)
    T = 1.0 # initial temperature
    alpha = 0.99 # temperature decay per iteration
    for i in range(its):
        xnew = nbr(x) # generate a neighbour of x
        fxnew = f(xnew)
        if (fxnew < fx or
            random() < exp((fx - fxnew) / T)):
            x = xnew
            fx = fxnew
        T *= alpha
    return x, fx
```

Simulated annealing

- Advantage: we can escape local optima. Performance is **much** better than simple hill-climbing.
- Disadvantage: hyperparameters.

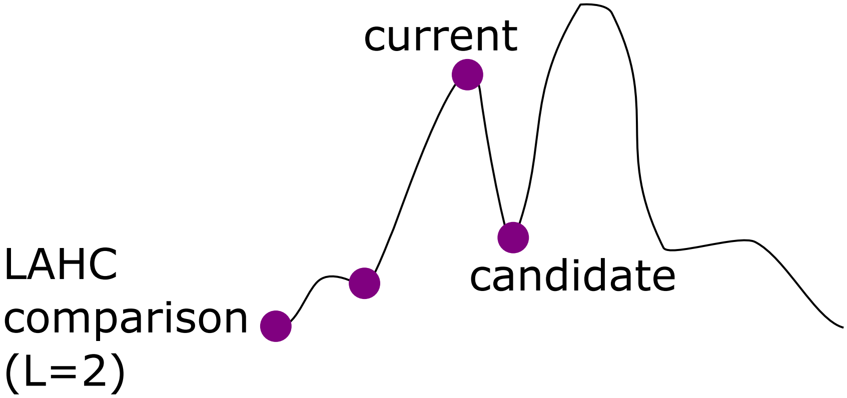
Nice explanation here

The big problem with simulated annealing is that the user has to think about the annealing schedule. What initial value should T have, and how quickly should it decrease? These are **hyperparameters**. SA might work well with one configuration of them, but badly with another. There's no configuration which works well for all problems. So, SA is harder to use than HC, according to Burke & Bykov (2008).

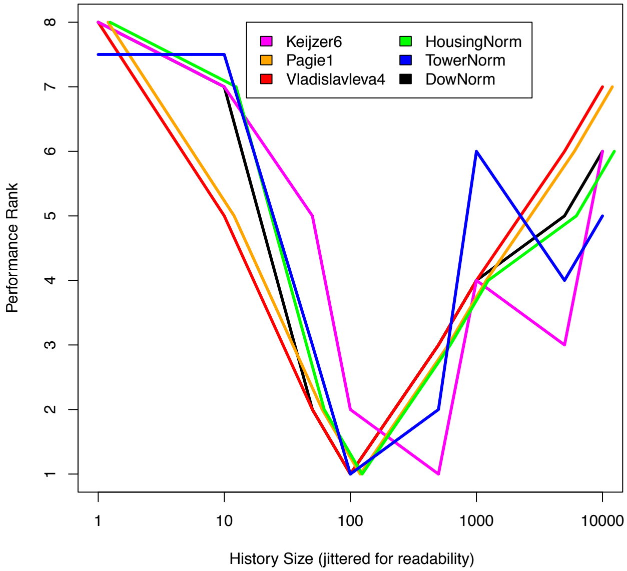
Late-acceptance hill-climbing

- LAHC is an alternative to SA, due to Burke & Bykov
- Again, hope to escape local optima by sometimes accepting disimproving moves
- Keep a *history* of recent `f` values, of length L
- Instead of accepting x' when it is better than x (as in hill-climbing),
- we accept x' when it is better than x_L, that is the x we had L steps previously: we make the acceptance decision **late**
- L the only hyperparameter, easy to control.

Late-acceptance hill-climbing



L is easy to control



```python
def LAHC(L, n, C, init, nbr):
    s = init()                # initial solution
    Cs = C(s)                 # cost of current solution
    best = s                  # best-ever solution
    Cbest = Cs                # cost of best-ever
    f = [Cs] * L              # initial history
    for I in range(1, n):     # number of iterations
        s_ = nbr(s)           # candidate solution
        Cs_ = C(s_)           # cost of candidate
        if Cs_ < Cbest:       # minimising
            best = s_         # update best-ever
            Cbest = Cs_
        v = I % L             # v indexes I circularly
        if Cs_ <= f[v] or Cs_ <= Cs:
            s = s_            # accept candidate
            Cs = Cs_          # (otherwise reject)
        f[v] = Cs             # update circular history
    return best, Cbest
```

Quiz

In LAHC, if we set $L = 1$, what happens?

- The algorithm breaks
- It behaves like a random walk
- It behaves like hill-climbing

Quiz

In LAHC, if we set $L = 1$, what happens?

- The algorithm breaks
- It behaves like a random walk
- It behaves like hill-climbing

Answer: it reduces to hill-climbing, because it always compares to the current best point, never to an old one.

Quiz

In LAHC, if we set L = `its`, what happens?

- The algorithm breaks
- It behaves like a random walk
- It behaves like hill-climbing

Quiz

In LAHC, if we set L = `its`, what happens?

- The algorithm breaks
- It behaves like a random walk
- It behaves like hill-climbing

Answer: it behaves like a random walk, because it never compares new points to anything.

Quiz

In LAHC, if we set L = `its`, what happens?

- The algorithm breaks
- It behaves like a random walk
- It behaves like hill-climbing

Answer: it behaves like a random walk, because it never compares new points to anything.

A random **walk** is not the same as random **search**. It is like hill-climbing where we **always** accept the move, i.e. there is no objective function.

Eg: https://upload.wikimedia.org/wikipedia/commons/c/cb/Random_wal

Quiz

In LAHC, increasing L leads to:

- More exploration
- More exploitation

Quiz

In LAHC, increasing L leads to:

- More exploration
- More exploitation

Answer: Larger L allows more disimproving moves, so more exploration.

Multiple points and information transfer

Our final option for improving on hill-climbing is to use **multiple search points** with **information transfer** between them.

This leads to the idea of a **genetic algorithm**. We'll study this next week.

Suggested readings

Two research papers in Blackboard – not required reading:

- Kirkpatrick et al. (SA)
- Burke & Bykov (LAHC for exam timetabling)