Settling for Good Solutions

Week 11

COMP 1201 (Algorithmics)

ECS, University of Southampton

22 May 2020

Previously...

Backtracking, Branch and Bound

- **Backtracking**: a powerful technique for solving constraint problems with large state spaces.
- Can take an exponential amount of time, but a good implementation will often find solutions relatively quickly.
- Delivers a huge improvement over exhaustive search for solutions in a large search space.
- Can be used to solve intractable problems in practice.
- **Branch and Bound**: a widely applicable technique for solving discrete optimisation problems.
- Similar to Backtracking – using cost as a constraint to prune away chunks of the state space.

Heuristics

Given that there are currently no known efficient algorithms for finding optimal solutions to **NP-Hard** problems, we are left with:

1. spending potentially a very long time searching for an optimal solution (e.g. using Branch and Bound), or
2. accepting "good" solutions which aren't necessarily optimal.

Algorithms for finding good solutions are often called approximation algorithms or **heuristic algorithms**.

The idea behind heuristic algorithms is to use a rough guide or **heuristic** pointing us in a reasonable direction.

Heuristics

If a heuristic is good, it will help us find good solutions much faster than exhaustive search.

Two commonly used heuristics are:

1. A greedy heuristic (take the best move available).
2. Believe that good solutions are “close” to each other.

Heuristic algorithms fall broadly into two categories:

1. Constructive algorithms.
2. Local (neighbourhood) search.

Constructive Algorithms

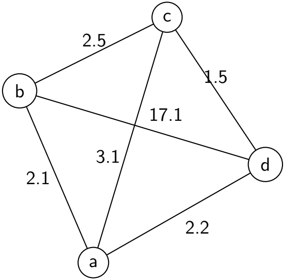
- Constructive algorithms build up a solution from scratch.
- They usually rely on a greedy heuristic.
- They are very fast.
- Once they obtain a solution, they stop.
- They can give reasonable solutions very quickly, but the quality of the solution is often not very good.

Constructive Algorithms

- Constructive algorithms build up a solution from scratch.
- They usually rely on a greedy heuristic.
- They are very fast.
- Once they obtain a solution, they stop.
- They can give reasonable solutions very quickly, but the quality of the solution is often not very good.

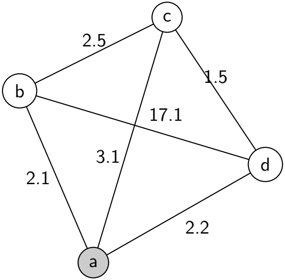
Example: greedy heuristic for the Travelling Salesman Problem.

Constructive Algorithms (Greedy Heuristic)



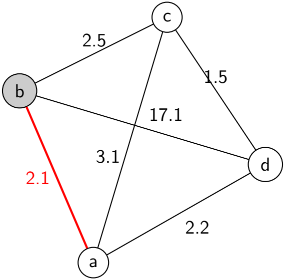
TSP: find the shortest Hamiltonian tour in a complete finite graph.

Constructive Algorithms (Greedy Heuristic)



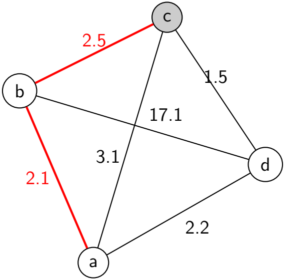
Step 1: pick a vertex.

Constructive Algorithms (Greedy Heuristic)



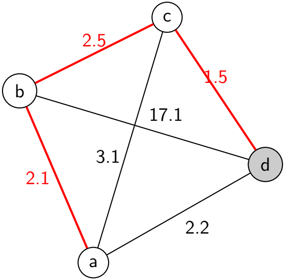
Step 2: visit the nearest unvisited vertex.

Constructive Algorithms (Greedy Heuristic)



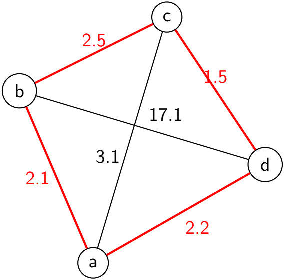
Step 2: (repeat) visit the nearest unvisited vertex.

Constructive Algorithms (Greedy Heuristic)



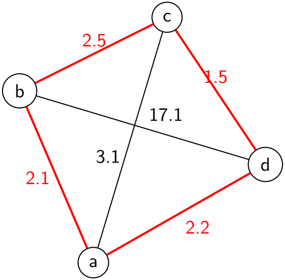
Step 2: (repeat) visit the nearest unvisited vertex.

Constructive Algorithms (Greedy Heuristic)



Step 2: (repeat) visit the nearest unvisited vertex.

Constructive Algorithms (Greedy Heuristic)



Step 3: **stop** when a solution is obtained.

Neighbourhood Search

An alternative to constructive algorithms are neighbourhood search (local search) algorithms.

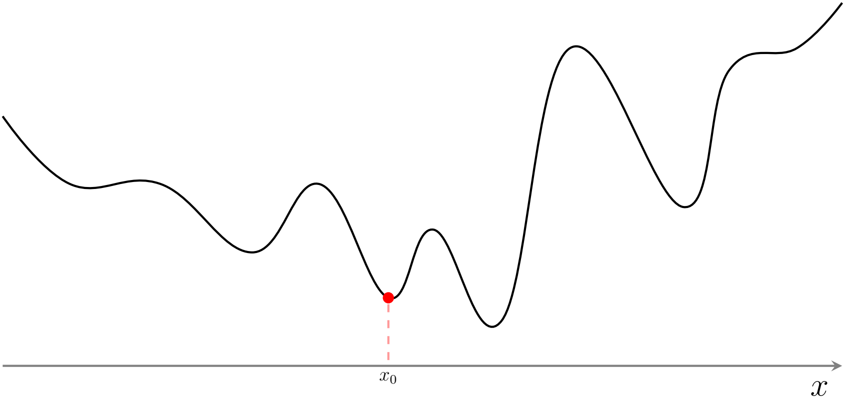
In neighbourhood search we:

1. Start from some initial solution.
2. Examine the neighbouring solutions.
3. Move to a neighbour if it is better (or simply not worse).
4. Repeat step 2 until some stopping criterion is met.

If we are maximising, this strategy is known as a **hill-climber**.

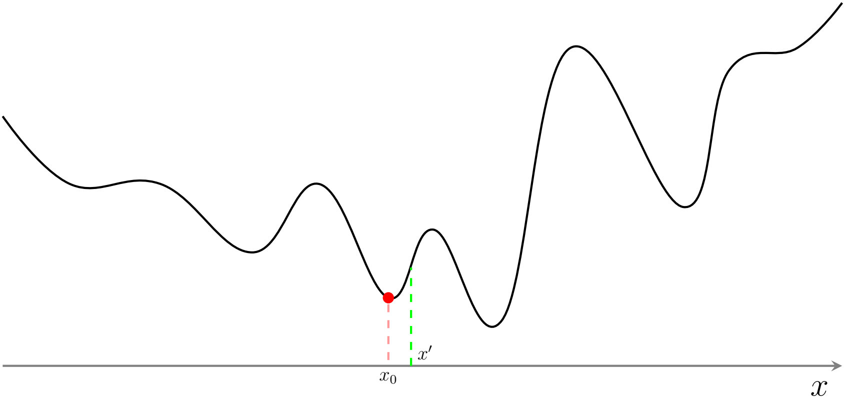
If we are minimising, this is often called **descent**.

Neighbourhood Search



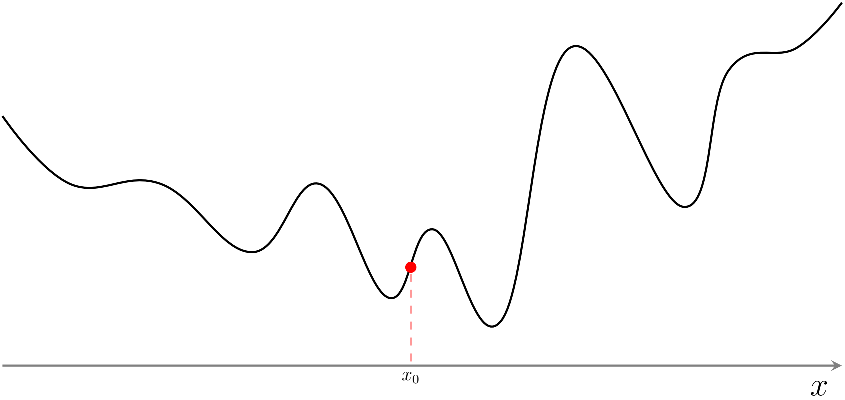
Hill-climbing: start at some initial solution x_0.

Neighbourhood Search



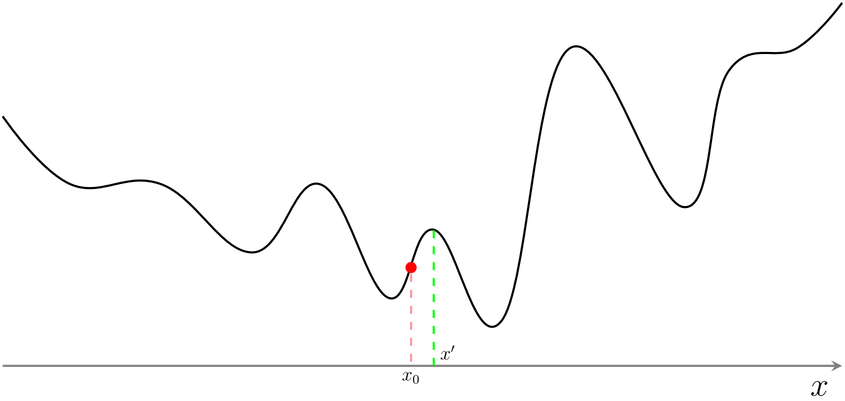
Hill-climbing: consider a neighbouring solution x'.

Neighbourhood Search



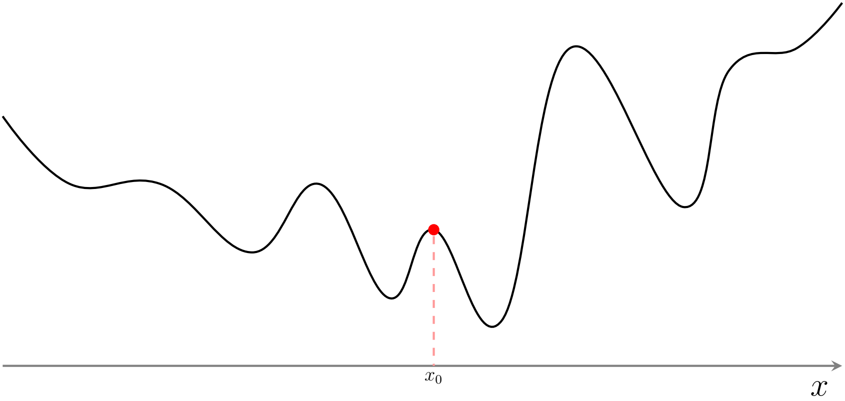
Hill-climbing: if the neighbouring solution is better, move to it.

Neighbourhood Search



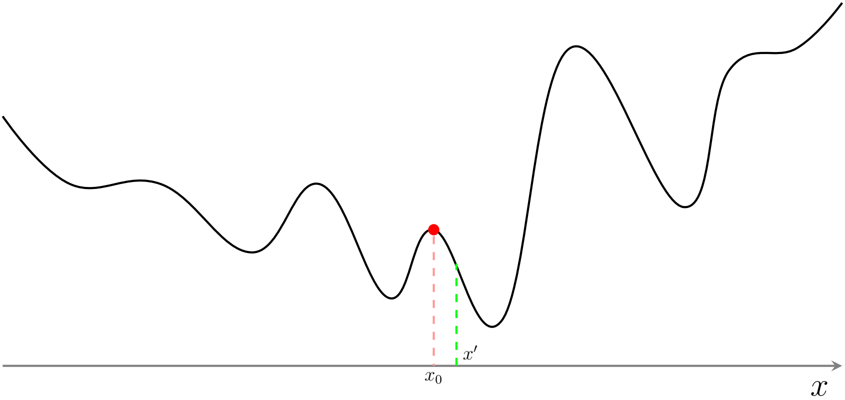
Hill-climbing: (repeat) consider an new neighbouring solution x'.

Neighbourhood Search



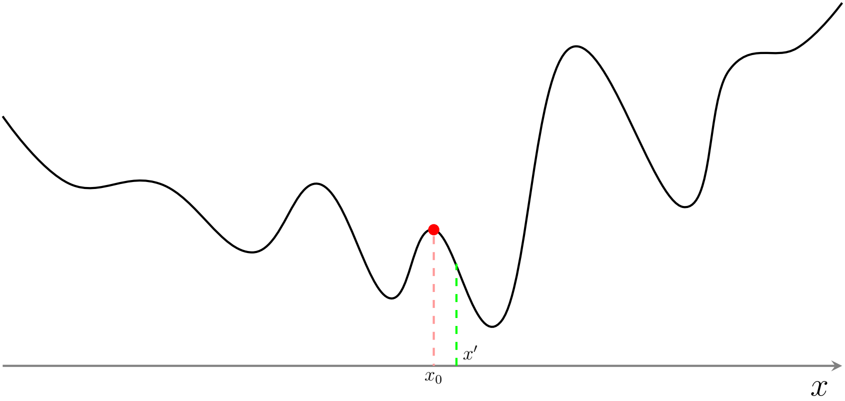
Hill-climbing: if the neighbouring solution is better, move to it.

Neighbourhood Search



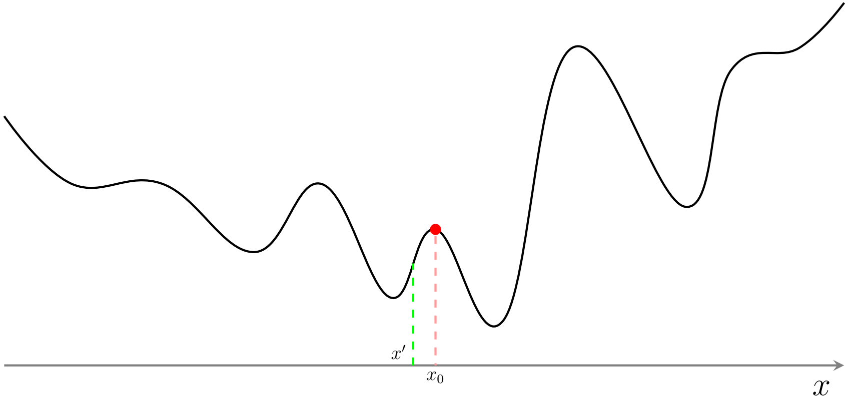
Hill-climbing: (repeat) consider an new neighbouring solution x'.

Neighbourhood Search



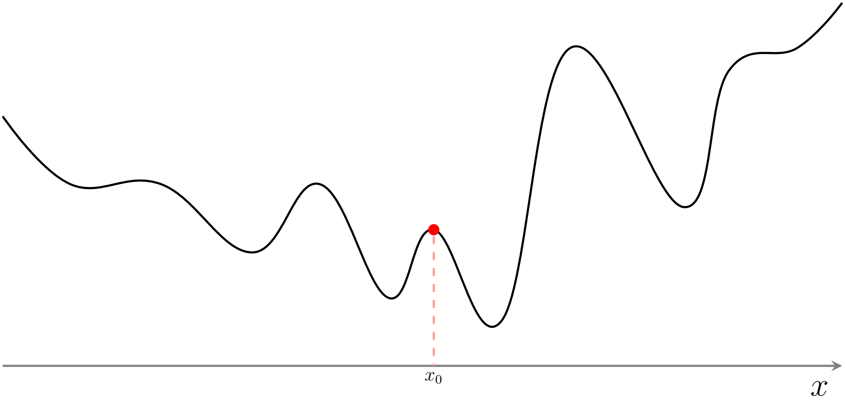
Hill-climbing: (repeat) consider an new neighbouring solution x'.
If it is not an improvement, look for another neighbour.

Neighbourhood Search



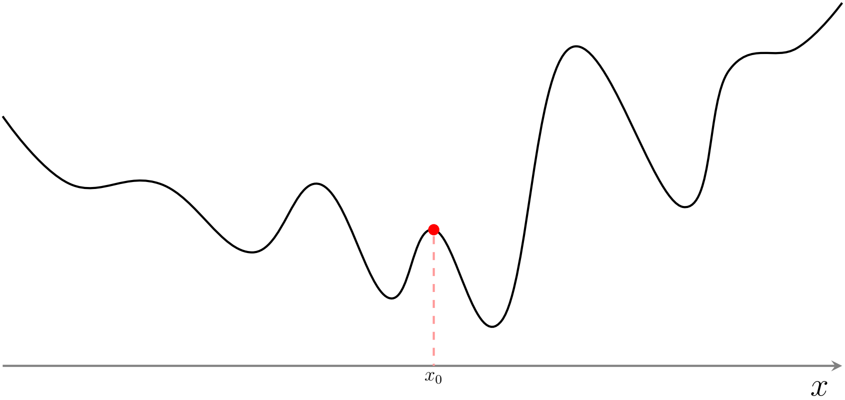
Hill-climbing: (repeat) consider an new neighbouring solution x'.
If it is not an improvement, look for another neighbour.

Neighbourhood Search



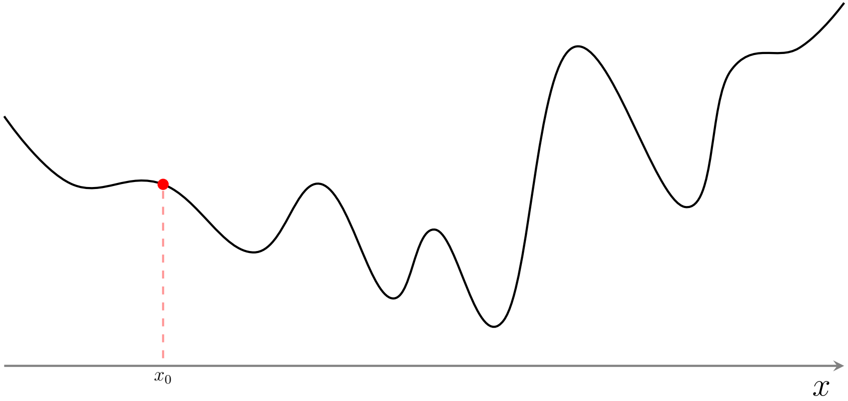
Hill-climbing: **stop** when there are no better neighbouring solutions.

Neighbourhood Search



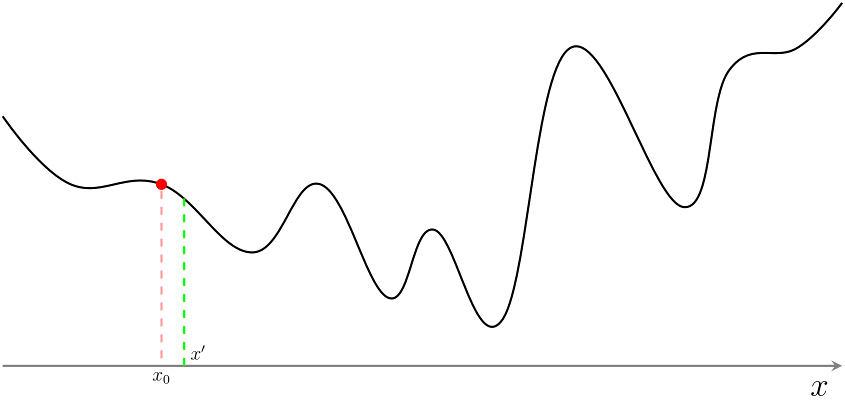
Hill-climbing: **stop** when there are no better neighbouring solutions. We have found a *(local) maximum*.

Neighbourhood Search



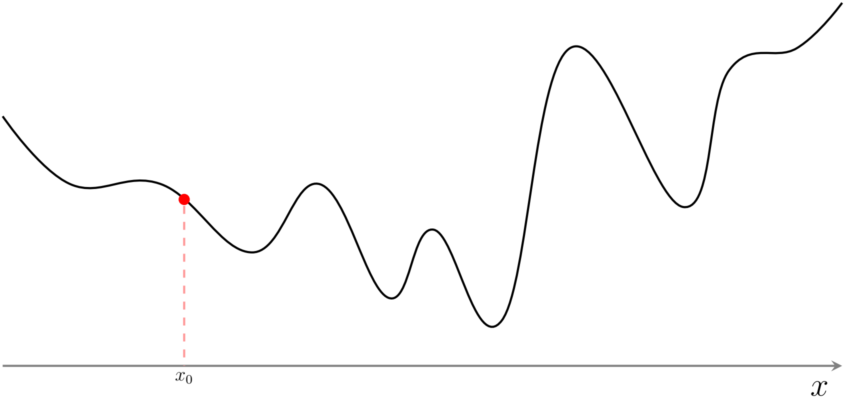
Descent: start at some initial solution x_0.

Neighbourhood Search



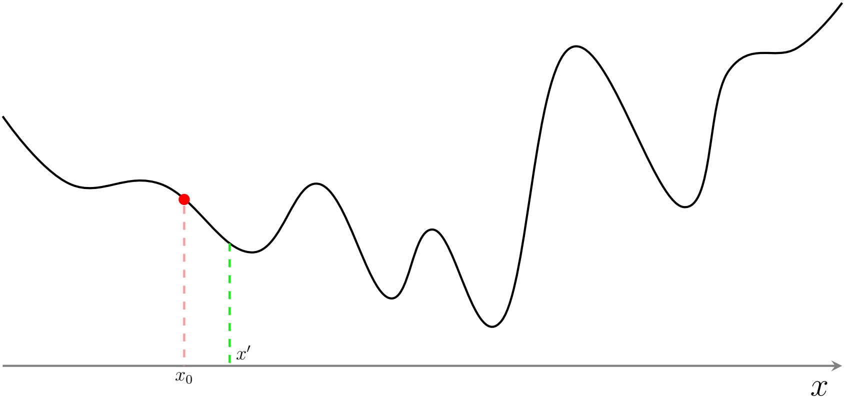
Descent: consider a neighbouring solution x'.

Neighbourhood Search



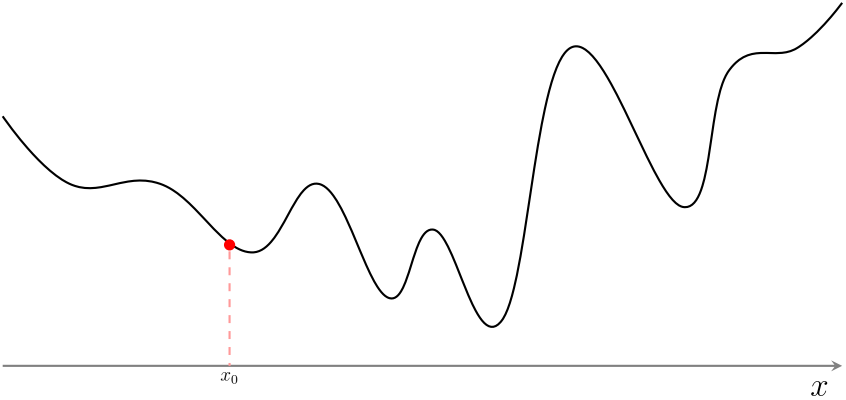
Descent: if the value of the objective function is lower at x', move there.

Neighbourhood Search



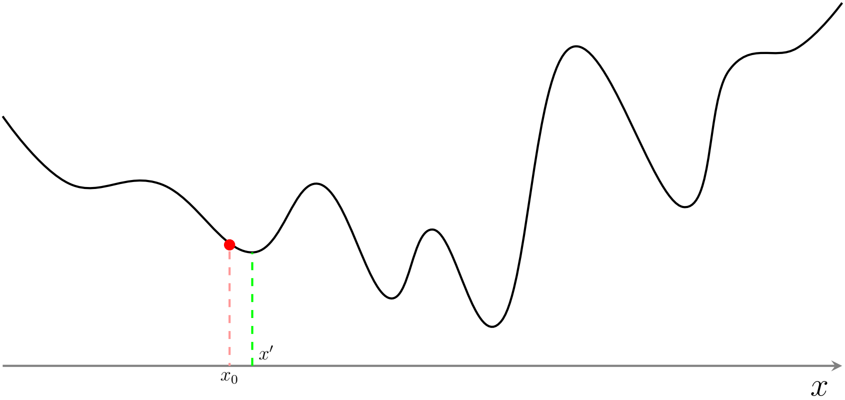
Descent: (repeat) consider a new neighbouring solution x'.

Neighbourhood Search



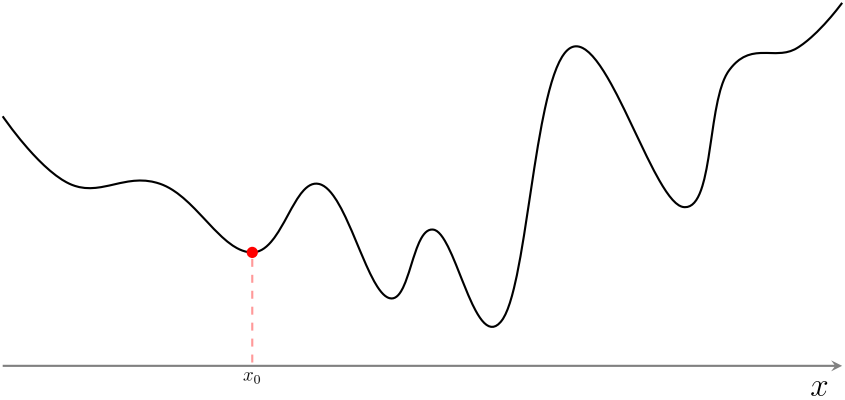
Descent: if the value of the objective function is lower at x', move there.

Neighbourhood Search



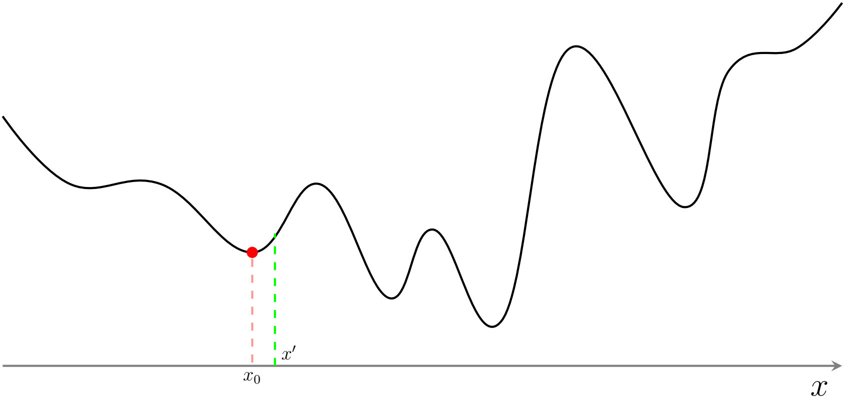
Descent: (repeat) consider a new neighbouring solution x'.

Neighbourhood Search



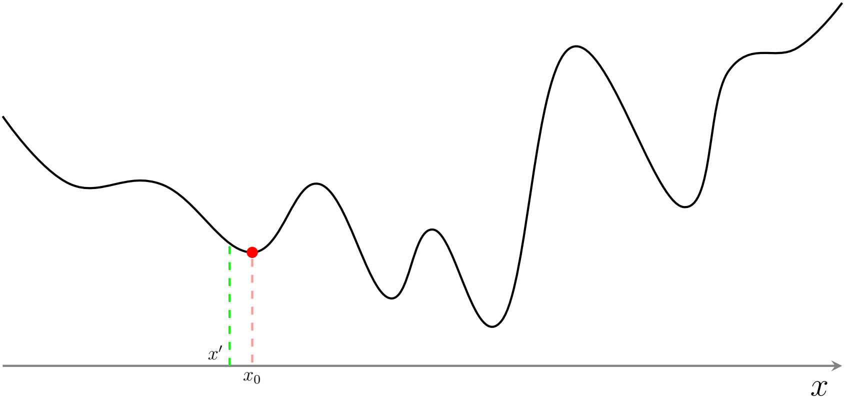
Descent: if the value of the objective function is lower at x', move there.

Neighbourhood Search



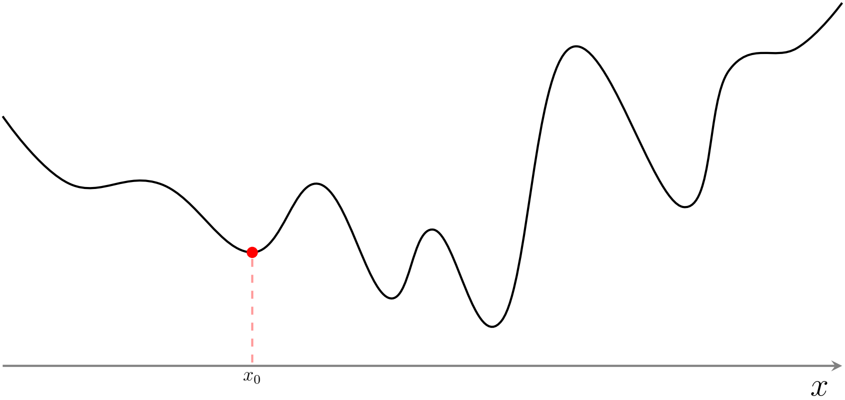
Descent: (repeat) consider another neighbouring solution x'.

Neighbourhood Search



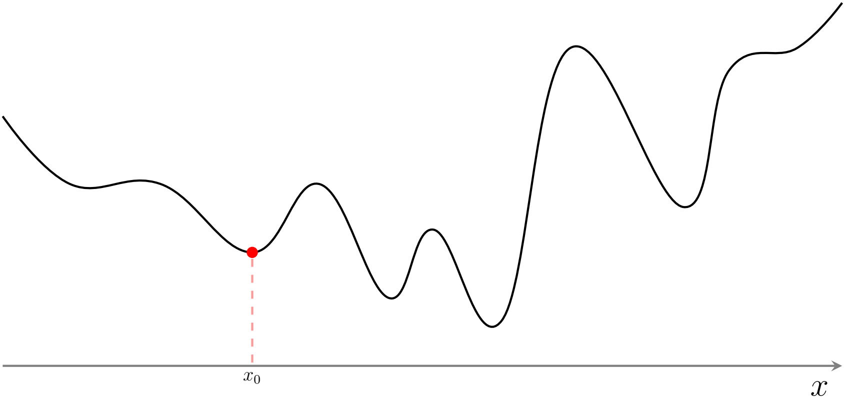
Descent: if it is worse, look for another neighbour x'.

Neighbourhood Search



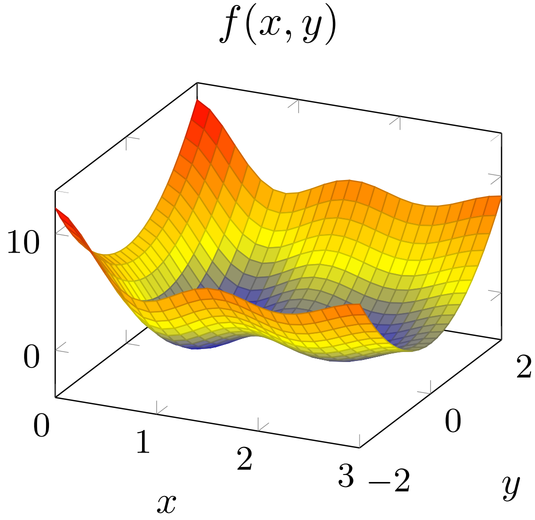
Descent: **stop** when there are no neighbouring solutions with lower values of the objective function.

Neighbourhood Search



Descent: **stop** when there are no neighbouring solutions with lower values of the objective function. We are at a *(local) minimum*.

Higher-dimensional State Spaces



Iterative Improvement at its Best

- There are times when a neighbourhood search algorithm will find the global optimum.
- A classic example of this is in Linear Programming where the Simplex method finds the global optimum.
- Unfortunately, this does not always work because many optimisation problems are *non-convex* (i.e. a local optimum is not necessarily the global optimum).
- Neighbourhood search is usually much slower than a constructive algorithm, but tends to find better quality solutions.
- However, it will often <u>get stuck</u> at a local optimum.

Simple Fixes

- One very simple fix is to restart neighbourhood search from many different starting positions.
- We could also easily perturb the current solution, and restart.
- These are good improvements over doing nothing, but aren't necessarily great strategies.
- We can also increase the size of our neighbourhood when selecting neighbours to decrease the chance of getting stuck. For instance, in TSP we could swap more cities in our solution.

Simple Fixes

- One very simple fix is to restart neighbourhood search from many different starting positions.
- We could also easily perturb the current solution, and restart.
- These are good improvements over doing nothing, but aren't necessarily great strategies.
- We can also increase the size of our neighbourhood when selecting neighbours to decrease the chance of getting stuck. For instance, in TSP we could swap more cities in our solution.
- However, we can do something more sophisticated...

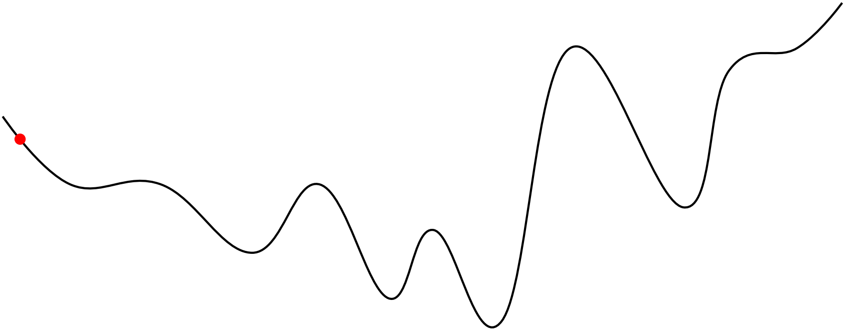
Simulated Annealing

- Simulated Annealing is an example of a stochastic hill-climber method. It first received serious interest in the 1980s.
- Sometimes you go in the wrong direction (i.e. down hill).
- It is named in analogy to physical annealing:

  A crystalline solid is heated and then left to slowly cool until it is free of crystal defects (i.e. reaches its lowest crystal lattice energy state).

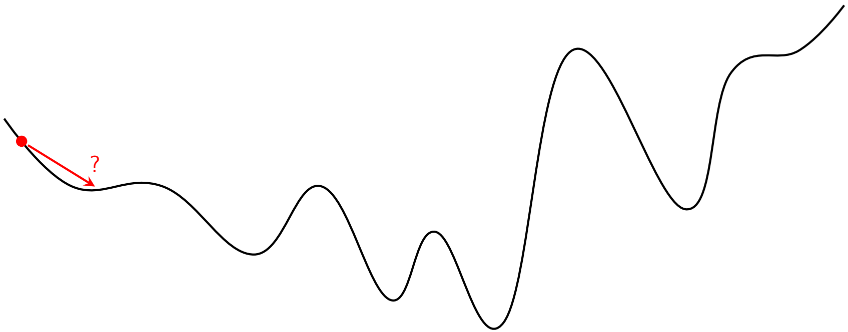
- Simulated Annealing applies this idea to search for global minima in discrete optimisation problems.

Stochastic Descent



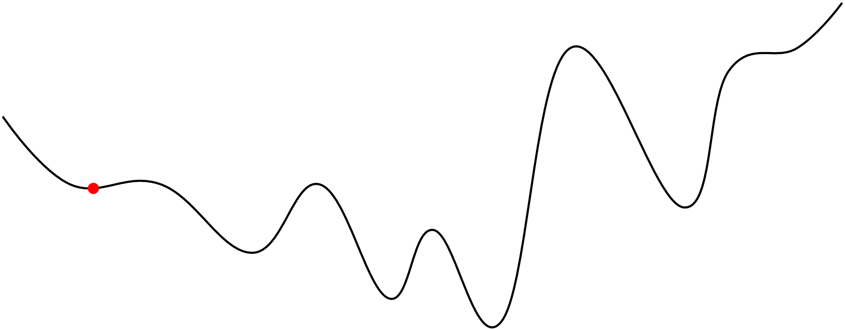
Physical intuition: It is easier to fall down hill than to go back up.

Stochastic Descent



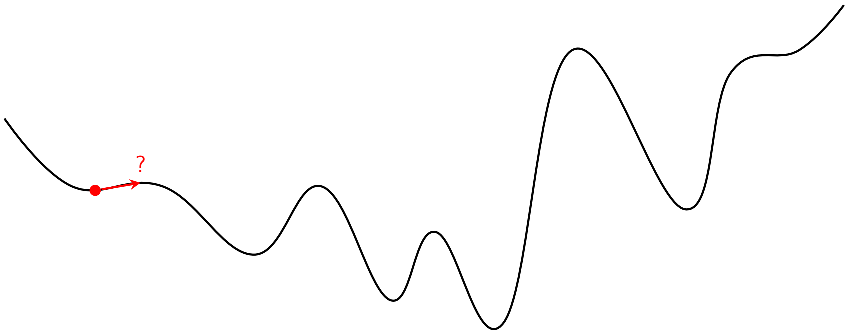
Physical intuition: It is easier to fall down hill than to go back up.

Stochastic Descent



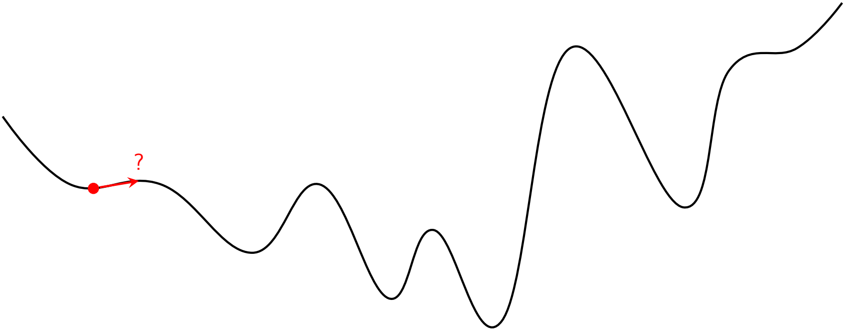
Physical intuition: It is easier to fall down hill than to go back up.

Stochastic Descent



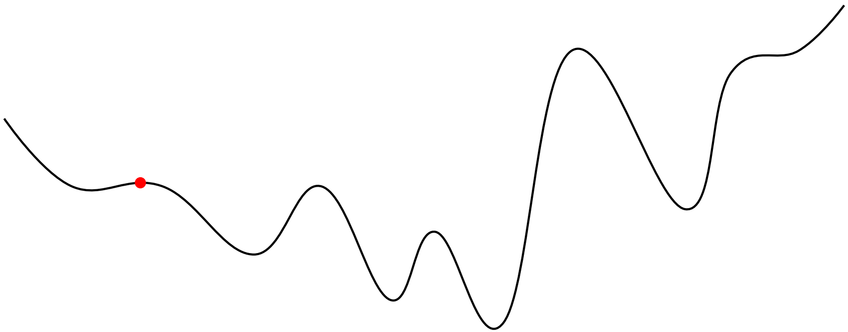
We don't need to improve our solution at every step.

Stochastic Descent



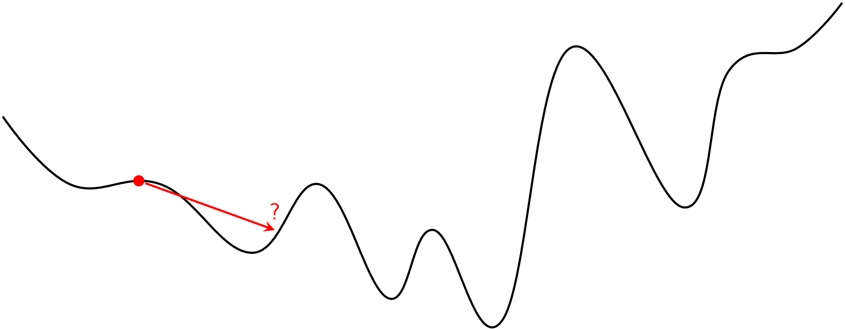
We can make "bad moves" with some probability.

Stochastic Descent



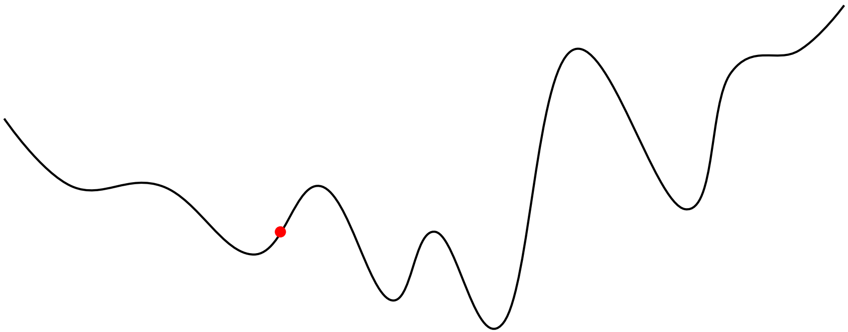
We can make “bad moves” with some probability.

Stochastic Descent



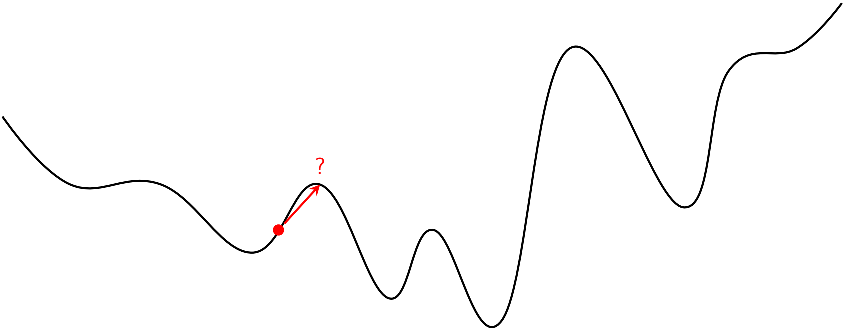
We can make "bad moves" with some probability.

Stochastic Descent



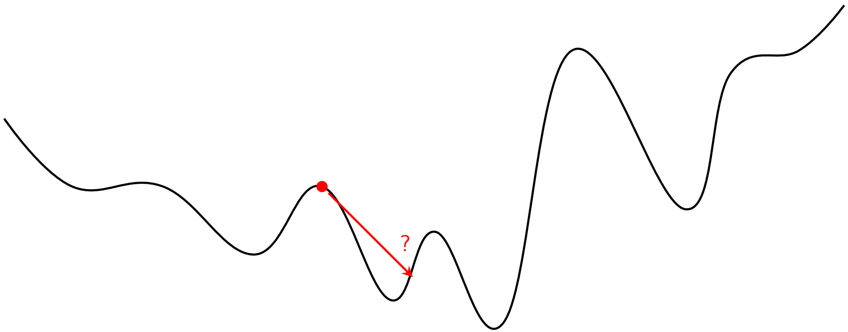
When there are “good moves” available, we should take those.

Stochastic Descent



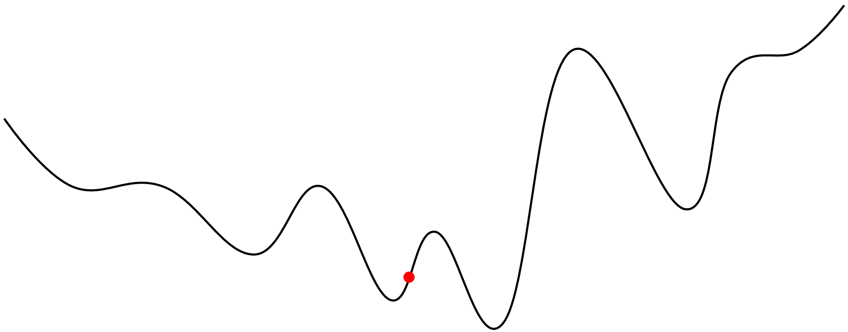
We have a chance of getting out of “troughs” and “valleys”

Stochastic Descent



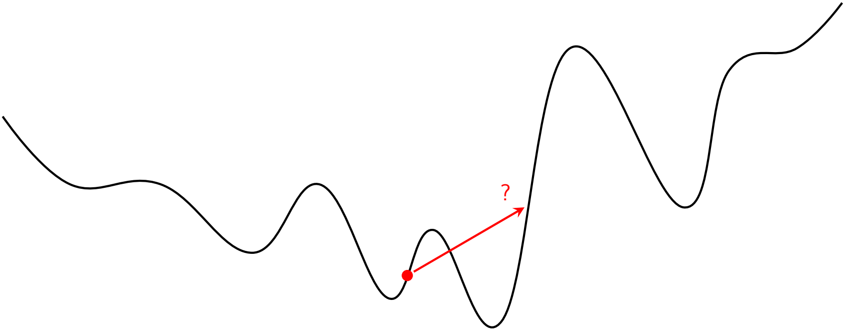
This affords us the possibility of finding better optima.

Stochastic Descent



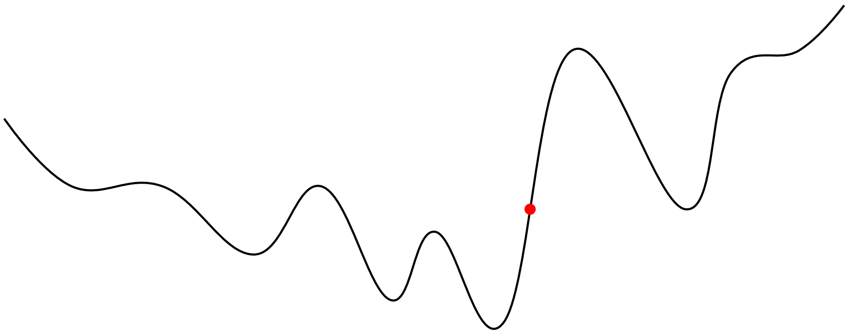
This affords us the possibility of finding better optima.

Stochastic Descent



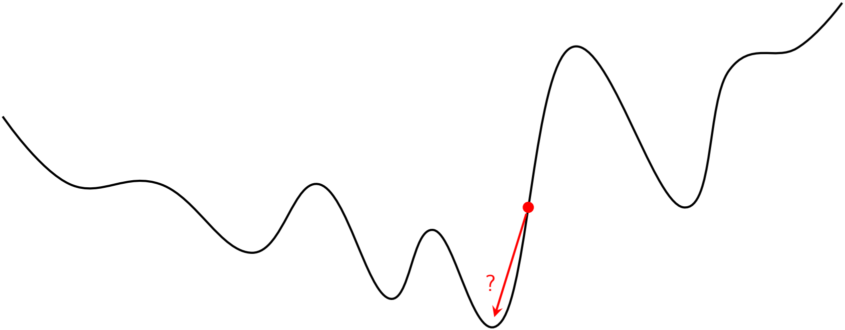
This affords us the possibility of finding better optima.

Stochastic Descent



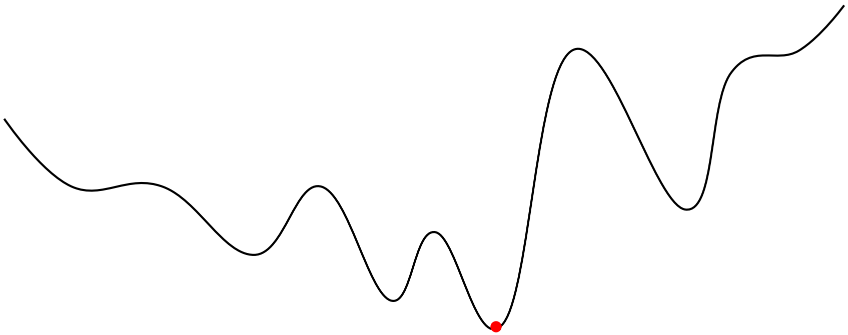
This affords us the possibility of finding better optima.

Stochastic Descent



This affords us the possibility of finding better optima.

Stochastic Descent



This affords us the possibility of finding better optima.

Simulated Annealing

Algorithm to **minimise energy** $E(\vec{X})$, where $\vec{X} = (X_1, X_2, \ldots, X_n)$.

- Start from a <u>random initial configuration</u> $\vec{X}$,
- Choose a neighbour $\vec{X}'$,
- If the neighbour is better (has lower energy), <u>move to it</u>,
- Otherwise, move to the neighbour <u>with some probability</u>.
- A parameter β controls the probability of moving to the neighbour.
- We increase β to reduce the probability of going uphill over time.

Cooling Schedule

- The parameter β is known as the inverse temperature because of an analogy with physics.
- Over time, we have to increase β (i.e. decrease temperature) so that the system will remain in a low energy state.
- The way you reduce the temperature (increase β) is known as the **cooling schedule**.
- Choosing a good cooling schedule can be critical.
- Choosing a good cooling schedule is something of a black art.

Convergence Theorem

- There is a theorem that says if you choose a slow enough cooling schedule you will end up in the global optimum <u>eventually</u>.
- Unfortunately 'eventually' is a very long time.
- It is quicker to search through all possible states.
- Still, people get excited about convergence proofs.

Genetic Algorithms

Genetic Algorithms (GA)

- Inspired by the theory of evolution.
- Invented by **John Holland** in the 1960s.
- An influential book *Adaptation in Natural and Artificial Systems* by Holland is published in 1975.
- By 1980s Genetic Algorithms were being used in many areas.
- The term **Genetic Programming (GP)** was introduced by J. Koza in 1992 to refer to the use of Genetic Algorithms to evolve programs.

J. H. Holland

Genetic Algorithms

- Genetic Algorithms (GAs) are methods to evolve a *population of potential candidates* to find a good solution to an optimisation problem.
- GAs are part of a family of related methods known as **Evolutionary Algorithms (EAs)**.
- Can be viewed as a biologically-inspired "engineering approach" to solving hard problems.

Genetic Algorithms: Some Terminology

- Individual: any possible solution.
- Population: a set of individuals.
- Fitness: objective function that we are trying to optimise.
  [*Every individual can be evaluated according to this function.*]
- Chromosome: representation of a solution.
  [*Often chromosomes take the form of strings or vectors.*]
- Gene: a position (or a set of positions) in a chromosome.
- Allele: the possible values in a gene.

Genetic Algorithms: Some Terminology

- Individual: any possible solution.
- Population: a set of individuals.
- Fitness: objective function that we are trying to optimise.
  [*Every individual can be evaluated according to this function.*]
- Chromosome: representation of a solution.
  [*Often chromosomes take the form of strings or vectors.*]
- Gene: a position (or a set of positions) in a chromosome.
- Allele: the possible values in a gene.

E.g. chromosome of an individual with alleles A,B,...,G, could be:

ABBCDAFAGBBAAFGFCBCA

A Canonical GA

1. Initialise population.

A Canonical GA

1. Initialise population.
2. `for` $t = 1$ `to` T `do:`
   (a) Evaluate fitness of members of the population.

A Canonical GA

1. Initialise population.
2. `for` $t = 1$ `to` T `do:`
   (a) Evaluate fitness of members of the population.
   (b) Select a new population based on fitness.

A Canonical GA

1. Initialise population.
2. `for` $t = 1$ `to` T `do:`
   (a) Evaluate fitness of members of the population.
   (b) Select a new population based on fitness.
   (c) Mutate members of the population.

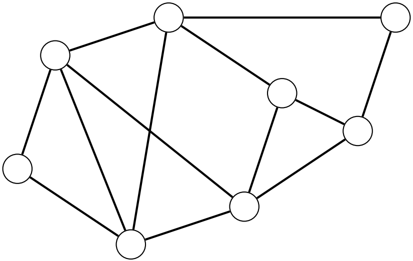
A Canonical GA

1. Initialise population.
2. `for` $t = 1$ `to` T `do:`
   (a) Evaluate fitness of members of the population.
   (b) Select a new population based on fitness.
   (c) Mutate members of the population.
   (d) Crossover members of the population.

A Canonical GA

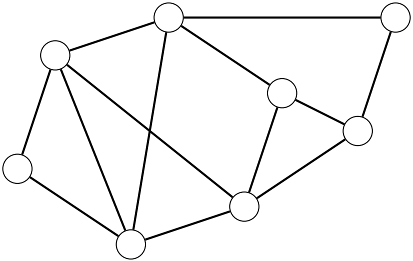
1. Initialise population.
2. `for` $t = 1$ `to` T `do:`
   (a) Evaluate fitness of members of the population.
   (b) Select a new population based on fitness.
   (c) Mutate members of the population.
   (d) Crossover members of the population.
3. Return the best member of the population.

Genetic Algorithms (Example: Graph Colouring)



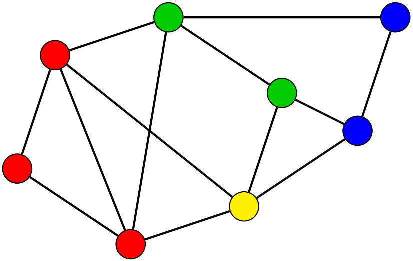
- Given a finite graph $G = (V, E)$.
- Assign colours, $c(v)$, to all the vertices of the graph, $v \in V$.
  [colours from a finite set, $c : V \to C$, where $|C| < |V| \in \mathbb{N}$.]
- Minimise the number of edges that connect vertices with the same colour, i.e. edges $e = (v, v') \in E$ such that $c(v) = c(v')$.

Genetic Algorithms (Example: Graph Colouring)



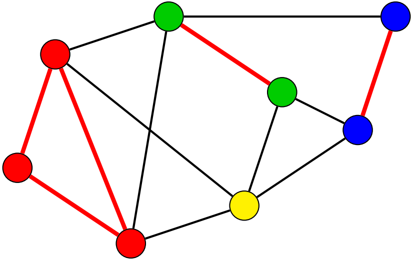
- Given a finite graph $G = (V, E)$. Let $C = \{R, G, B, Y\}$.
- Assign colours, $c(v)$, to all the vertices of the graph, $v \in V$.
  [colours from a finite set, $c : V \rightarrow C$, where $|C| < |V| \in \mathbb{N}$.]
- Minimise the number of edges that connect vertices with the same colour, i.e. edges $e = (v, v') \in E$ such that $c(v) = c(v')$.

Genetic Algorithms (Example: Graph Colouring)



- Given a finite graph $G = (V, E)$. Let $C = \{R, G, B, Y\}$.
- Assign colours, $c(v)$, to all the vertices of the graph, $v \in V$.
  [colours from a finite set, $c : V \to C$, where $|C| < |V| \in \mathbb{N}$.]
- Minimise the number of edges that connect vertices with the same colour, i.e. edges $e = (v, v') \in E$ such that $c(v) = c(v')$.

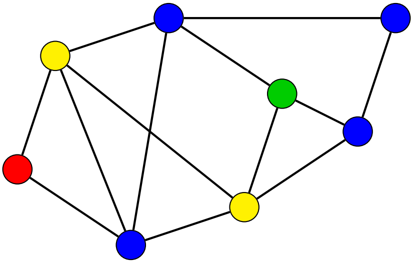
Genetic Algorithms (Example: Graph Colouring)



- Given a finite graph $G = (V, E)$. Let $C = \{R, G, B, Y\}$.
- Assign colours, $c(v)$, to all the vertices of the graph, $v \in V$. [colours from a finite set, $c : V \to C$, where $|C| < |V| \in \mathbb{N}$.]
- Minimise the number of edges that connect vertices with the same colour, i.e. edges $e = (v, v') \in E$ such that $c(v) = c(v')$.

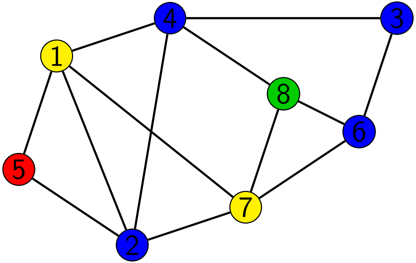
Step 1: Initialise Population

- An initial population may be created by generating *random graph colourings*.



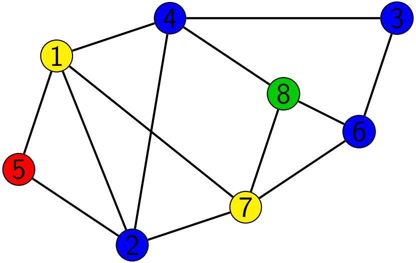
Step 1: Initialise Population

- An initial population may be created by generating *random graph colourings*.



Step 1: Initialise Population

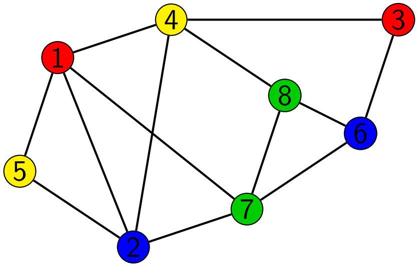
- An initial population may be created by generating *random graph colourings*.



{ YBBBRBYG }

Step 1: Initialise Population

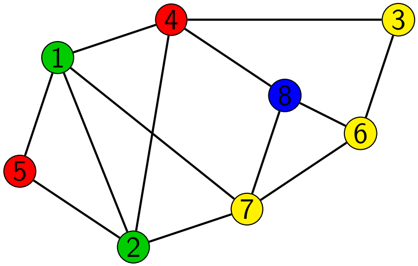
- An initial population may be created by generating *random graph colourings*.



{ YBBBRBYG, RBRYYBGG }

Step 1: Initialise Population

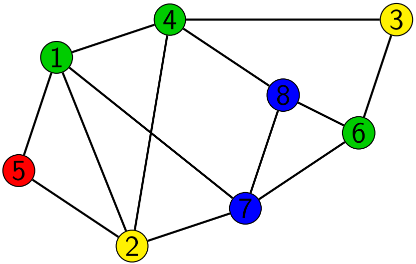
- An initial population may be created by generating *random graph colourings*.



{ YBBBRBYG, RBRYYBGG ,GGYRRYYB }

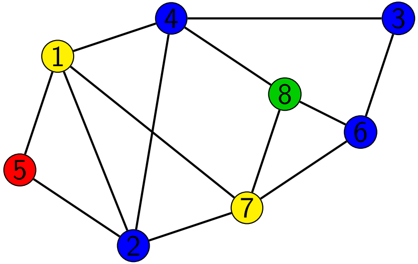
Step 1: Initialise Population

- An initial population may be created by generating *random graph colourings*.



{ YBBBRBYG, RBRYYBGG ,GGYRRYYB ,GYYGRGBB }

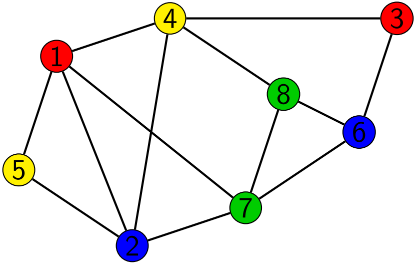
Step 2



(a) Evaluate fitness of individuals.

$$cost(YBBBRBYG) = 4$$

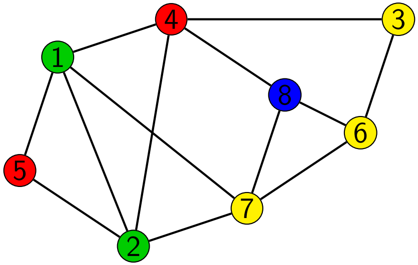
Step 2



(a) Evaluate fitness of individuals.

$$cost(YBBBRBYG) = 4,$$
$$cost(RBRYYBGG) = 1,$$

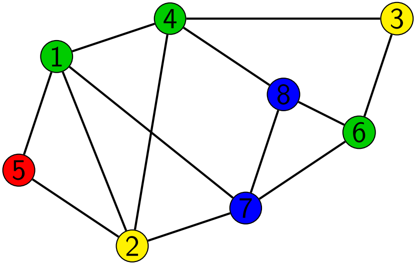
Step 2



(a) Evaluate fitness of individuals.

$$cost(YBBBRBYG) = 4,$$
$$cost(RBRYYBGG) = 1,$$
$$cost(GGYRRYYB) = 3,$$

Step 2



(a) Evaluate fitness of individuals.

$$cost(YBBBRBYG) = 4,$$
$$cost(RBRYYBGG) = 1,$$
$$cost(GGYRRYYB) = 3,$$
$$cost(GYYGRGBB) = 2.$$

Step 2

(b) Select a new population P of members preferentially choosing the fitter members.

- Let w_α be a measure of fitness of individual α.
- We could select members α with probability
  [N is the size of our population.]

$$p_\alpha = \frac{w_\alpha}{\sum_{\alpha'=1}^{N} w_{\alpha'}}$$

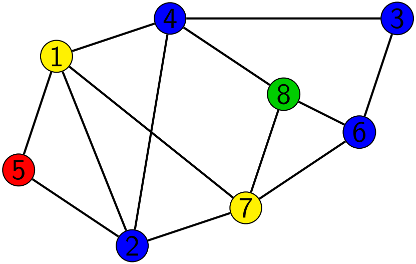
- This is known as **roulette wheel selection**.
- Many different ways of doing this.
- After selection, we should be left with a smaller but fitter population.

E.g. $P = \{$ RBRYYBGG, GYYGRGBB $\}$.

Step 2

(c) Mutate members of the population.

- Change the colour of one or more of the vertices.

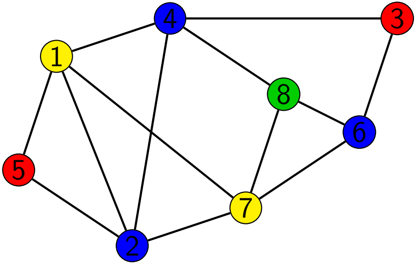


YBBBRBYG

Step 2

(c) Mutate members of the population.

- Change the colour of one or more of the vertices.

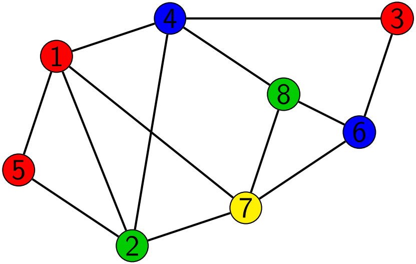


YBRBRBYG

Step 2

(c) Mutate members of the population.

- Change the colour of one or more of the vertices.

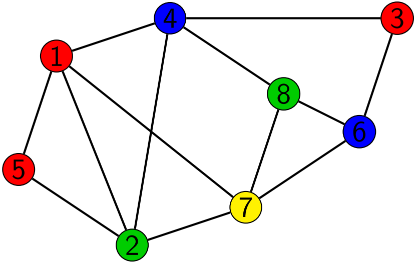


RGRBRBYG

Step 2

(d) Crossover members of the population.

- Take two solutions and combine them to form a new solution.



RGRBRBYG

Crossover Operators

- **Single-point crossover**: take two chromosomes, cut them at some *random site* and combine.

$$\left.\begin{array}{l} YRRB \mid GRRB \\ GGBR \mid RBGG \end{array}\right\} \longrightarrow \left\{\begin{array}{l} YRRB \mid RBGG \\ GGBR \mid GRRB \end{array}\right.$$

- **Multi-point crossover**: take two chromosomes and cut them at *several sites*, swapping alternating segments.

$$\left.\begin{array}{l} YRRB \mid GR \mid RB \\ GGBR \mid RB \mid GG \end{array}\right\} \longrightarrow \left\{\begin{array}{l} YRRB \mid RB \mid RB \\ GGBR \mid GR \mid GG \end{array}\right.$$

Crossover Operators

- **Uniform crossover**: take two chromosomes and create offspring by a *random shuffle*

$$\left.\begin{array}{l} YRRBGRRB \\ GGBRRBGG \end{array}\right\} \longrightarrow \left\{\begin{array}{l} YGBBRBGB \\ GRRRGRRG \end{array}\right.$$

- All of the above crossover operators can be biased towards one parent.
- **Bit-simulated crossover**: create offspring by choosing variables independently with probability proportional to the frequency of the allele in the population.
  E.g. $p(R) = 0.37, p(B) = 0.25, p(G) = 0.31, p(Y) = 0.06.$

Other Heuristics

- There are many extensions of Neighbourhood Search, Simulated Annealing, and Genetic Algorithms.
- There are also may other Evolutionary Algorithms (EAs):
  - Particle Swarm Optimisations (PSO),
  - Ant Colony Optimisation (ACO).
- Tabu search is another well-known search method.

Which Heuristic is Best?

- The best heuristic depends on the application.
- Descent is very fast, but only finds local optima – good starting place.
- Simulated Annealing and Genetic Algorithms are slow, but can often find good solutions.
- The best algorithms tend to be special purpose algorithms designed for the problem.

Further Reading:

Optional:

1. **A. E. Eiben, J. E. Smith**
   "*Introduction to Evolutionary Computing*"
   https://link.springer.com/book/10.1007/978-3-662-44874-8
   Companion website:
   http://www.evolutionarycomputation.org/

Acknowledgements: Partly based on earlier COMP 1201 slides by Drs Adam Prügel-Bennett, University of Southampton.