



Radio Russia

Wouter & Quinten



Inhoud

- Introductie
 - Het probleem
 - Constraints
 - State Space
 - Doelfunctie
- Theorie
- Methode
- Resultaten
- Discussie

Het Probleem

Het **doel**:
Een goede verdeling
van zendfrequenties.



Indeling zendfrequenties



Zeven types zendmasten



Vier kostenschema's



Geen overlap



5 kaarten

Oekraïne
USA
China
Russia
NL

Zenders

Zendertype	Kosten 1	Kosten 2	Kosten 3	Kosten 4
A	12	19	16	3
B	26	20	17	34
C	27	21	31	36
D	30	23	33	39
E	37	36	36	41
F	39	37	56	43
G	41	38	57	58

Het Probleem

Het **doel**:

Een goede verdeling
van zendfrequenties.

Constraints:

- Elke regio een zender
- Aangrenzende provincies niet dezelfde zendertypes

Eigenschappen:

- Elk zendertype heeft eigen kosten



Constraints

Hard Constraints:

- Geen twee zenders van hetzelfde type in regio's die een grens delen.
- Alle regio's moeten een zender toegekend krijgen.

Soft Constraints:

- Geen? -> Opties

Grens tussen
twee regio's

Zenders van
hetzelfde type



Doel

$$Kosten = \sum_{i=0}^n z_i \times k_i$$

Minimaliseren

n = #verschillende zendertypes

i = zendertypenummer

z_i = #zenders van type i

k_i = prijs van een zender van type i

Statespace USA

Eerste keer 7 opties

Staat er naast 6...

Daarnaast ??

Combinations and Permutations r: aantal keuzes n: aantal mogelijkheden per keuze		Repetition	
		yes ✓	no
Order	yes	n^r	$\frac{n!}{(n-r)!}$
	no ✓	$\frac{(r+n-1)!}{r!(n-1)!}$	$\frac{n!}{r!(n-r)!}$

Versimpel!

r = aantal staten = 50

n = aantal masten = 7

$$\frac{(50+7-1)!}{50!(7-1)!} = 32468436$$

State Space

Het aantal verdelingen van zendertypes over alle regio's

- Voor elke regio, elke unieke zendertype
- Geen rekening houdende met de constraints van burens
- Altijd alle regio's gevuld

Haken en ogen:

- Volgorde maakt niet uit?
- Kleuren Theorie

Formule:
$$\frac{(r+n-1)!}{r!(n-1)!}$$

r = aantal regio's

n = aantal zendertypes

Methoden

Constructief

- Random
- Greedy
- Depth First Search
- Breadth First Search

Iteratief

- Random re-assignment
- Hillclimber
- Simulated Annealing

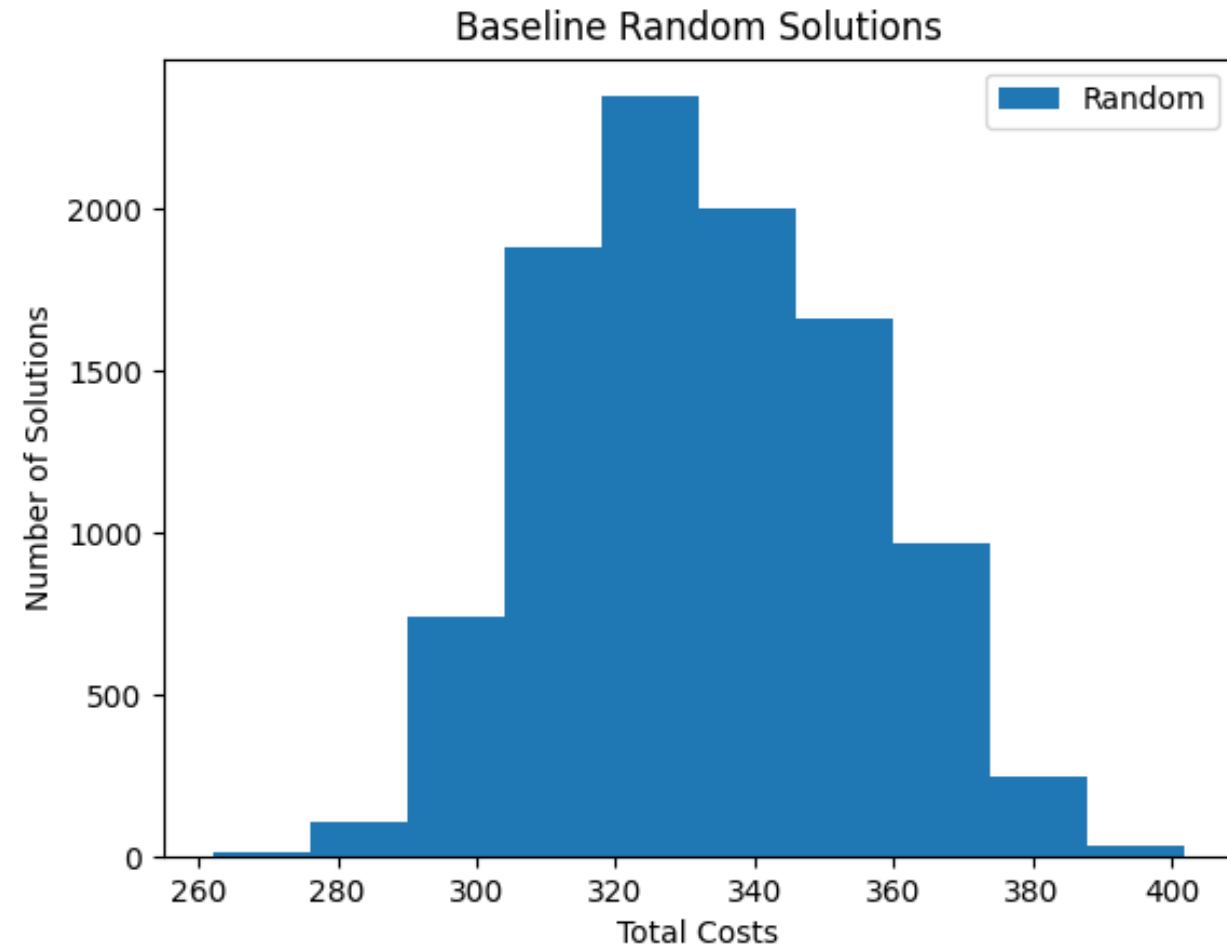
Random re-assignment

- Herhaal tot klaar of geen mogelijkheden:
 - Kies een node
 - Bekijk mogelijkheden voor transmitter
 - Assign random waarde

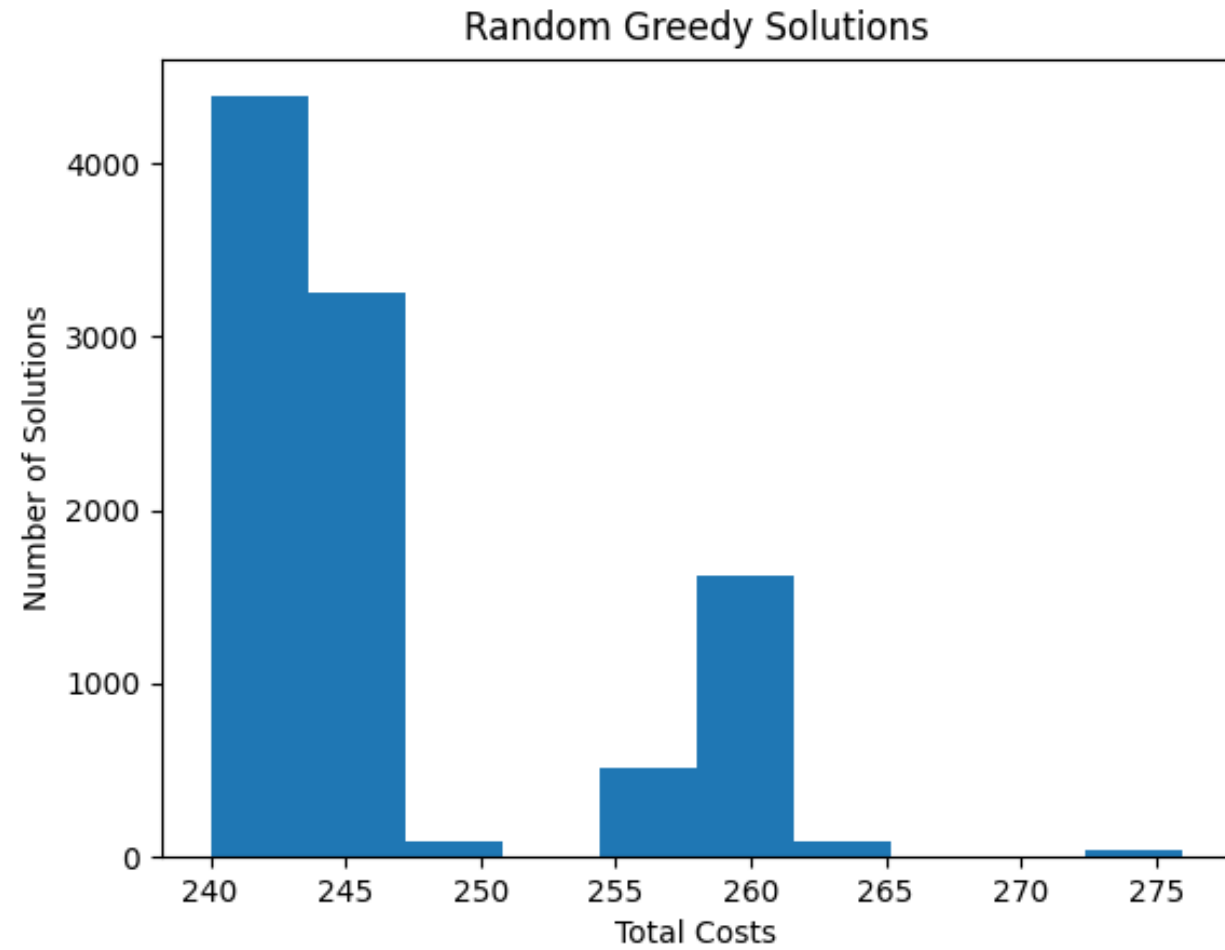
Greedy

- Herhaal tot klaar of geen mogelijkheden:
 - Kies een node (bijv. met meeste neighbours)
 - Bekijk mogelijkheden voor transmitter
 - Sorteer deze op waarde (laag -> hoog)
 - Assign de waarde

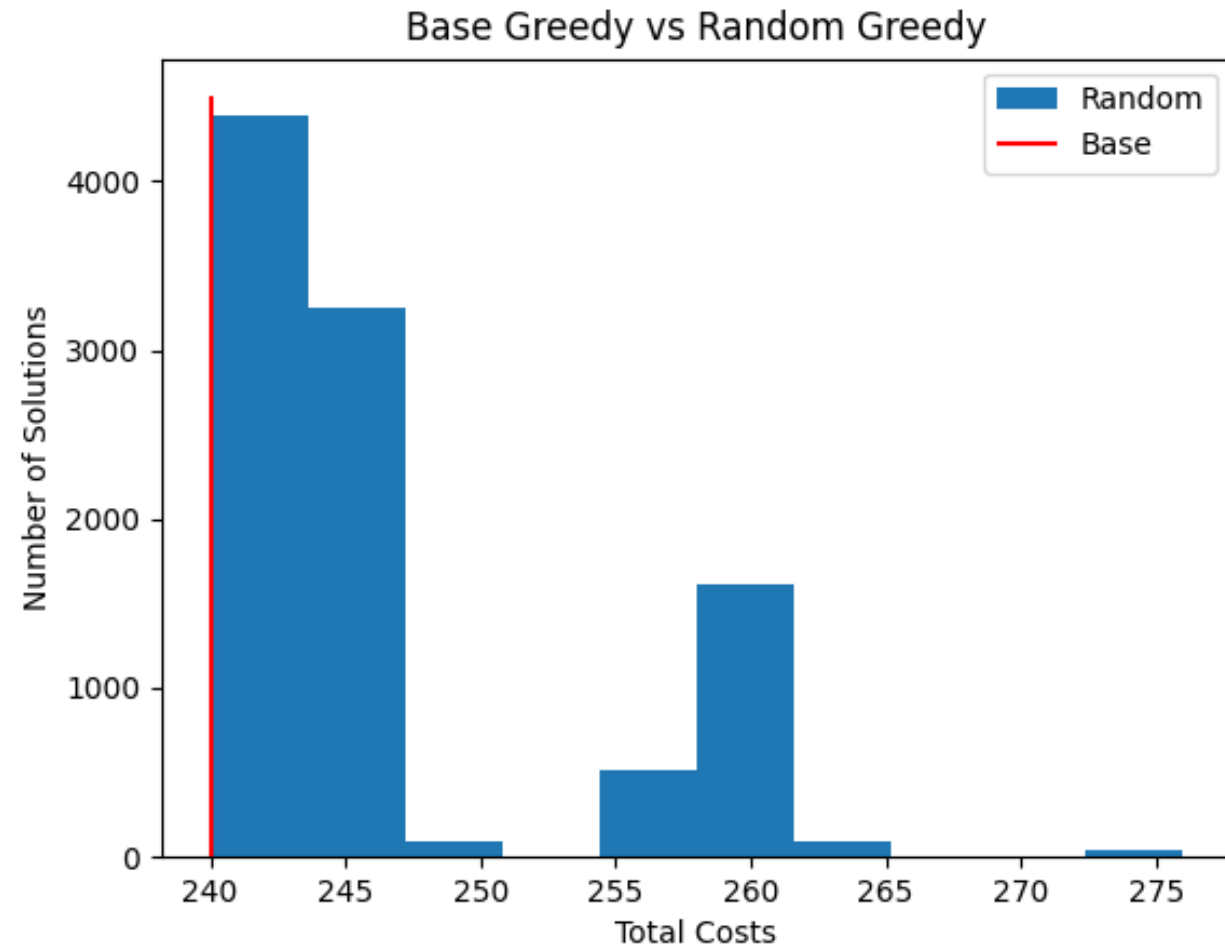
Constructieve Algoritmes - Random



Constructive Algorithms — Greedy



Constructieve Algoritmes — Greedy





Conclusie

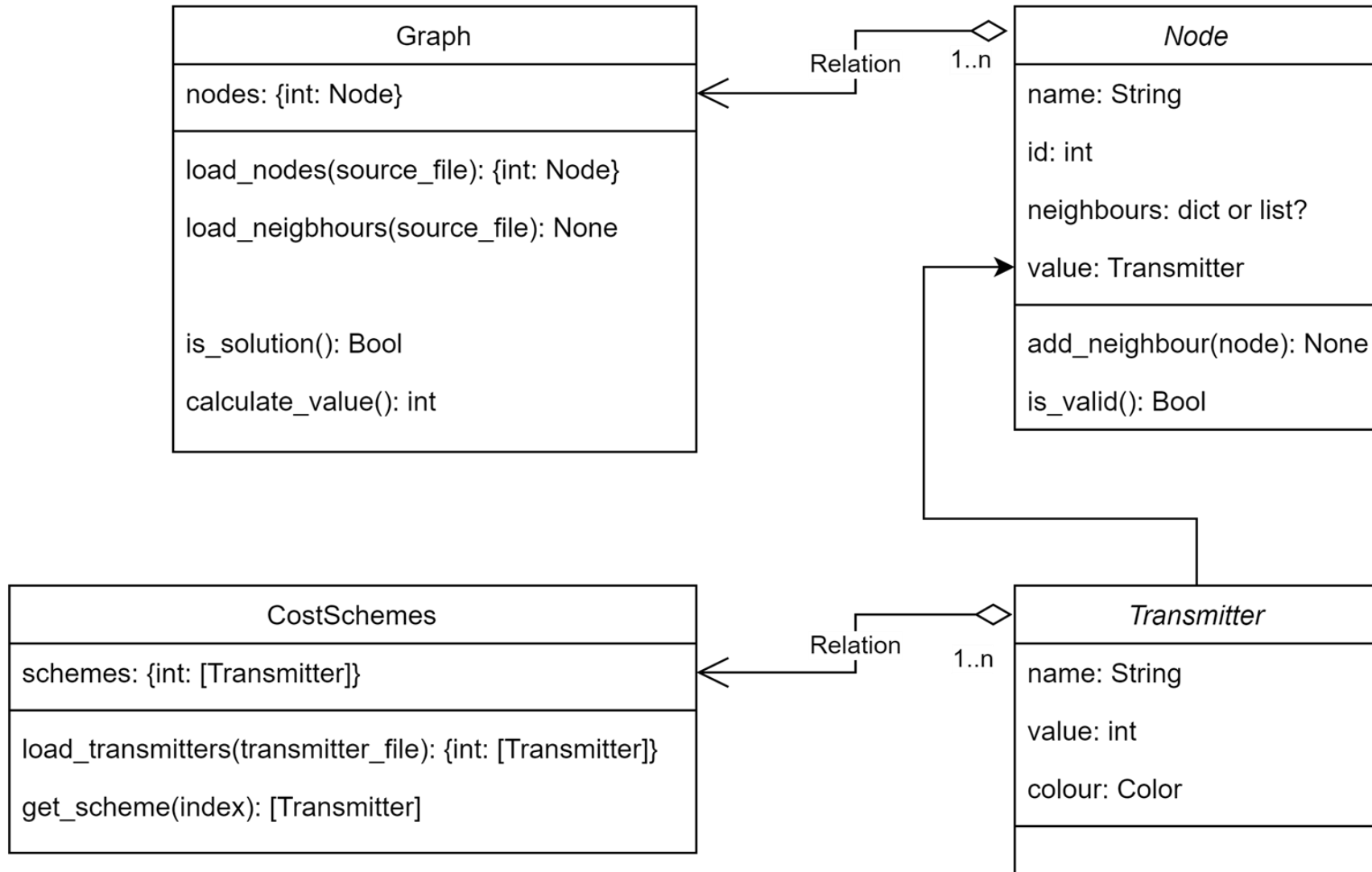
Algoritmen

Quinten van der Post & Wouter Vrieling

Kaart USA



Wat hebben we nu?



Wat hebben we nu?

```
def random_assignment(graph, possibilities):  
    """  
    Randomly assign each node with one of the possibilities.  
    """  
    for node in graph.nodes.values():  
        node.set_value(random.choice(possibilities))
```

Random re-assignment

- Herhaal tot klaar of geen mogelijkheden:
 - Kies een node
 - Bekijk mogelijkheden voor transmitter
 - Assign random waarde


```
...def get_violations(self):  
...    """  
...    Returns the ids of all nodes that have  
...    """  
...    violations = []  
...  
...    for node in self.nodes.values():  
...        if not node.is_valid():  
...            violations.append(node)  
...  
...    return violations
```

```
...def is_valid(self):  
...    """  
...    Returns whether the node is valid. A node is valid when there are no  
...    neighbours with the same value, and it's value is not None.  
...    """  
...    if not self.has_value():  
...        return False  
...  
...    for neighbour in self.neighbours.values():  
...        if neighbour.value == self.value:  
...            return False  
...  
...    return True
```

```

def random_reassignment(graph, possibilities):
    """
    Algorithm that reassigns nodes that are invalid until each node is valid.
    CAUTION: may run indefinitely.
    """
    new_graph = copy.deepcopy(graph)

    # Randomly assign a value to each of the nodes
    random_assignment(new_graph, possibilities)

    # Find nodes that are "violations" and have neighbours with same value
    violating_nodes = new_graph.get_violations()

    # While we have violations
    while len(violating_nodes):
        # Reconfigure violations randomly
        random_reconfigure_nodes(new_graph, violating_nodes, possibilities)

        # Find nodes that are violations
        violating_nodes = new_graph.get_violations()

    return new_graph

```

```

def random_assignment(graph, possibilities):
    """
    Randomly assign each node with one of the possibilities.
    """
    for node in graph.nodes.values():
        node.set_value(random.choice(possibilities))

```

```

def random_reconfigure_node(graph, node, possibilities):
    """
    Takes a node and assigns each node with one of the possibilities.
    """
    node.set_value(random.choice(possibilities))

def random_reconfigure_nodes(graph, nodes, possibilities):
    """
    Takes a list of nodes and assigns each node with one of the possibilities.
    """
    for node in nodes:
        random_reconfigure_node(graph, node, possibilities)

```

Algorithmen

Greedy

Depth First

Hill Climber

Greedy

- Herhaal tot klaar of geen mogelijkheden:
 - Kies een node (bijv. met meeste neighbours)
 - Bekijk mogelijkheden voor transmitter
 - Sorteer deze op waarde (laag -> hoog)
 - Assign de waarde

```
class Greedy:
    """
    The Greedy class that assigns the best possible value to each node one by one.
    """
    def __init__(self, graph, transmitters):
        self.graph = copy.deepcopy(graph)
        self.transmitters = transmitters
```

```

def run(self):
    """
    Greedily assigns the lowest costing transmitters to the nodes of the graph.
    """
    nodes = list(self.graph.nodes.values())

    node_possibilities = self.transmitters

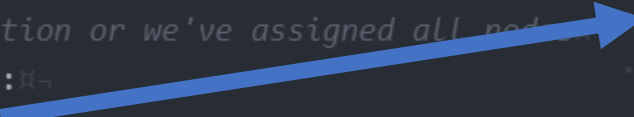
    # Repeat untill no more possible solution or we've assigned all nodes
    while nodes or not node_possibilities:
        node = self.get_next_node(nodes)

        # Retrieve all valid possible values for a node
        node_possibilities = node.get_possibilities(self.transmitters)

        # Sort them by value in ascending order
        node_possibilities.sort(key=lambda transmitter: transmitter.value)

        # Assign the lowest value possibility to the node
        node.set_value(node_possibilities[0])

```



```

def get_next_node(self, nodes):
    """
    Gets the next node with the most neighbours and removes it from the list.
    """
    nodes.sort(key=lambda node: len(node.neighbours))
    return nodes.pop()

```


Depth First

- Maak lege stack
- Voeg eerste staat toe op stack
- Herhaal tot stack leeg is:
 - Pak bovenste staat op de stack
 - Kies eerstvolgende node
 - Als we een node hebben:
 - Creëer nieuwe staten voor elk mogelijke keuze voor node
 - Anders:
 - Klaar?

Breadth ~~Depth~~ First

- Maak lege ~~stack~~ **queue**
- Voeg eerste staat toe op ~~stack~~ **queue**
- Herhaal tot stack leeg is:
 - Pak bovenste staat op de ~~stack~~ **queue**
 - Kies eerstvolgende node
 - Als we een node hebben:
 - Creëer nieuwe staten voor elk mogelijke keuze voor node
 - Anders:
 - Klaar?

Hill Climber

- Kies een random (geldige) staat:
- Herhaal x iteraties:
 - Maak een kopie van de huidige staat
 - Muteer de kopie
 - Als de staat is verbeterd
 - Vervang oude staat door nieuwe

Advanced stuff

Geavanceerdere Algoritmen



Genetische algoritmes
(PBA)

Differential Evolution
Plant Propagation
Fireworks



Ant Colony Optimisation



Particle Swarm Optimisation



Metropolis/Markov Chain Monte Carlo



Simulated Annealing

Demo
