Lecture 09 – Constructive Heuristics

Optimisation CT5141

James McDermott

University of Galway

OLLSCOIL NA GAILLIMHE

UNIVERSITY OF GALWAY

Overview

- **Opening the black box (a little)**
- A constructive heuristic for knapsack problems
- GRASP

From black-box to gray-box

"I would argue that the only way for evolutionary algorithms to be competitive on many classic NP Hard combinatorial optimization problems is to abandon 'Black Box' optimization and adopt more intelligent search methods." (Whitley, "A gray-box manifesto", in Bb).

From black-box to gray-box

"I would argue that the only way for evolutionary algorithms to be competitive on many classic NP Hard combinatorial optimization problems is to abandon 'Black Box' optimization and adopt more intelligent search methods." (Whitley, "A gray-box manifesto", in Bb).

The right response to this is that we should not use black-box methods **when the problem is not black-box**.

From black-box to gray-box

"I would argue that the only way for evolutionary algorithms to be competitive on many classic NP Hard combinatorial optimization problems is to abandon 'Black Box' optimization and adopt more intelligent search methods." (Whitley, "A gray-box manifesto", in Bb).

The right response to this is that we should not use black-box methods **when the problem is not black-box**.

This week we will see that some problems which we **could** address using black-box metaheuristics could instead be addressed by smarter methods.

And we'll see how to integrate the two.

Constructive heuristics

A **constructive heuristic** is a totally different approach to optimisation!

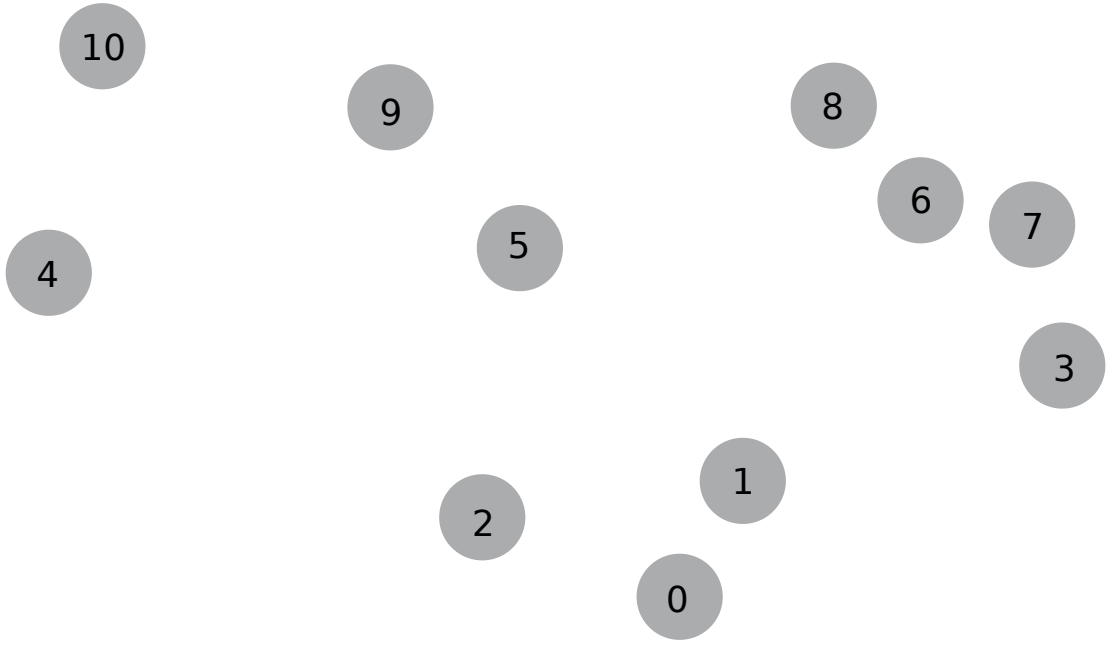
- **Metaheuristic search**: create complete but **poor-quality** solutions at random, then improve using selection, mutation, etc.

Constructive heuristics

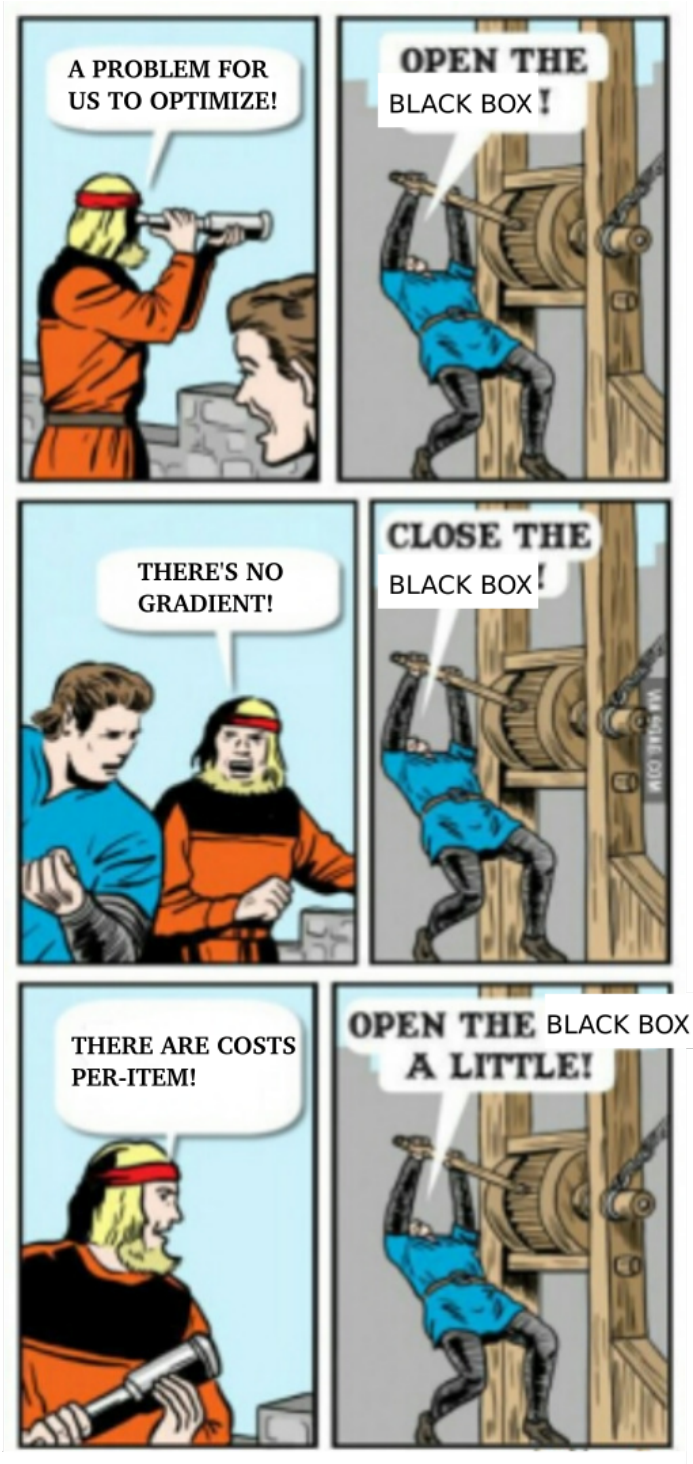
A **constructive heuristic** is a totally different approach to optimisation!

- **Metaheuristic search**: create complete but **poor-quality** solutions at random, then improve using selection, mutation, etc.
- **Constructive heuristic**: create an empty solution, then gradually build it up making **smart** choices at each step until complete.

A greedy heuristic for TSP: nearest city next

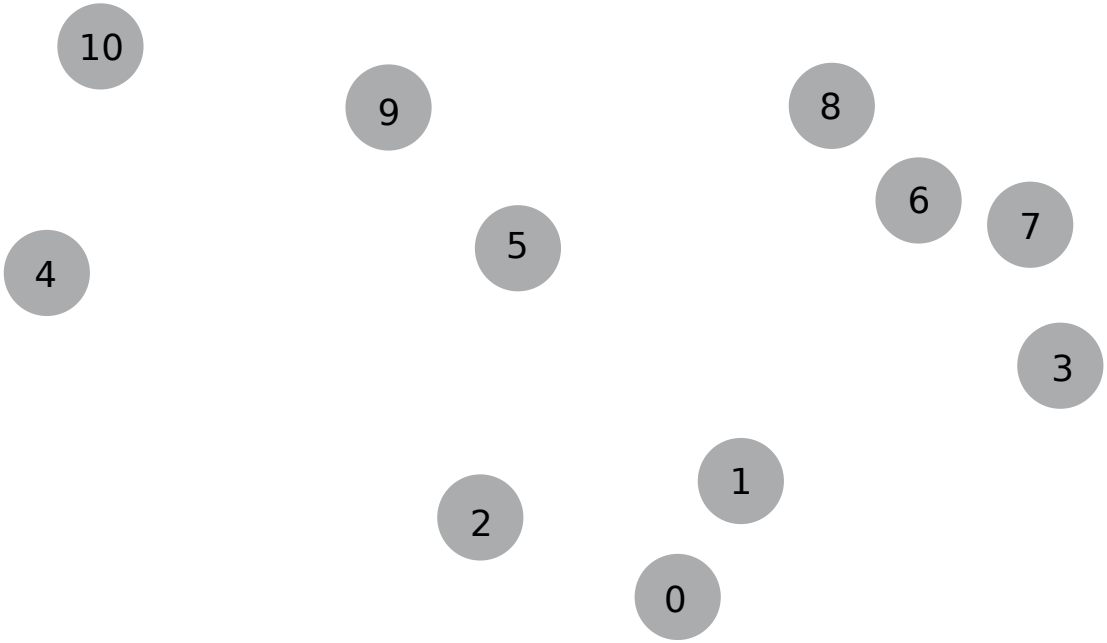


1. Start at City 0
2. Find the **nearest unvisited city**
3. Move to it, and mark it as visited.
4. If unvisited cities remain, go to 2.


A PROBLEM FOR US TO OPTIMIZE!
OPEN THE BLACK BOX!
THERE'S NO GRADIENT!
CLOSE THE BLACK BOX!
THERE ARE COSTS PER-ITEM!
OPEN THE BLACK BOX A LITTLE!

Greedy algorithms

- A **greedy** algorithm is one that makes the best choice it can right now, ignoring the future
- Sometimes, a greedy algorithm gives a sub-optimal result
- After several **nearest city** choices, it might find it has to take a very long step next
- Start at city 0 below and use the greedy algorithm.



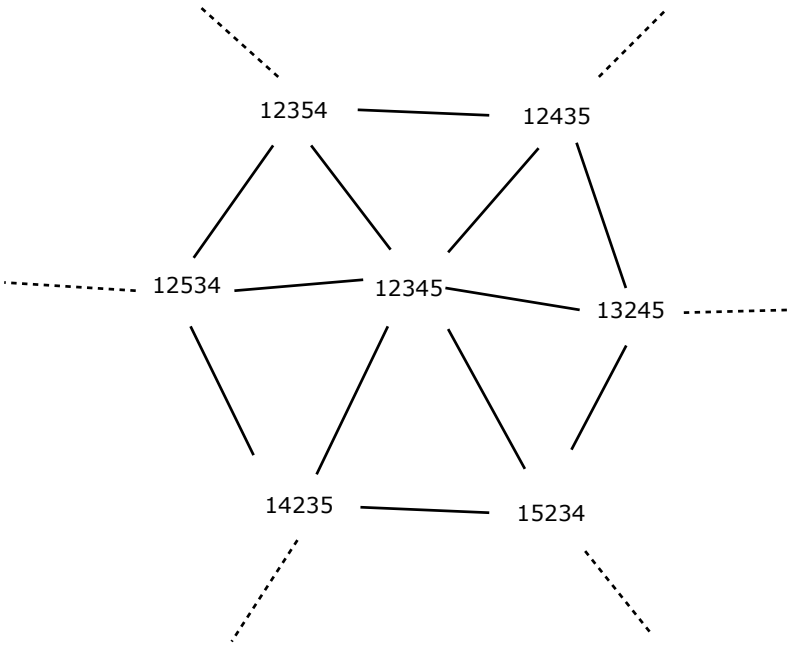
Greedy algorithms

- We probably achieve a solution that is much better than random
- But probably not optimal.

Deterministic and non-deterministic algorithms

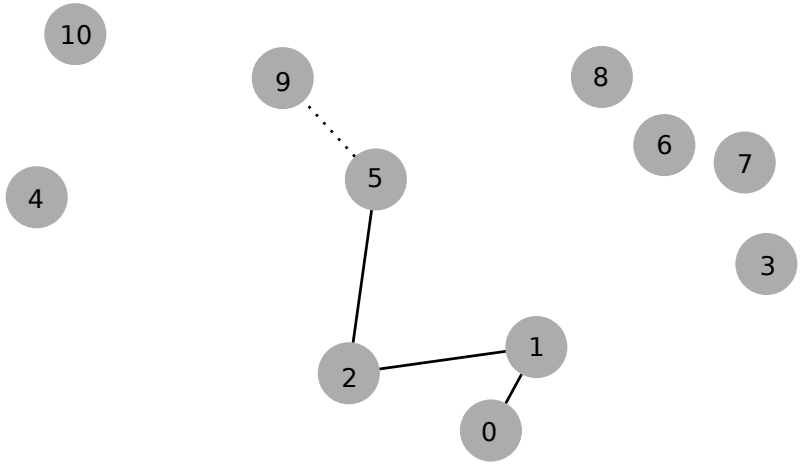
- Our greedy algorithm is deterministic: no random choices
- Non-deterministic versions exist, e.g. start at a random city and then proceed as before
- Do multiple runs and hope that at least one will give us a **better** result.

Reminder



Metaheuristic search

- Search space: permutations of n items
- $X = \mathbb{P}^n$



Constructive heuristic

- Same search space
- No neighbour operator, so no landscape, no local optima
- (Still no guarantee we find the global optimum)

Overview

- Opening the black box (a little)
- **A constructive heuristic for knapsack problems**
- GRASP

Application: Keyword advertising

Suppose we are a clothing retail website and we want to advertise our products via Google searches. When Google users search for terms such as “rain jackets”, we would like our ad to appear.

Application: Keyword advertising

Suppose we are a clothing retail website and we want to advertise our products via Google searches. When Google users search for terms such as “rain jackets”, we would like our ad to appear.

The profit for a keyword depends on its cost, the number of searches which are expected to contain that keyword, and on the **click-through rate** for that keyword.

Application: Keyword advertising

Suppose we are a clothing retail website and we want to advertise our products via Google searches. When Google users search for terms such as “rain jackets”, we would like our ad to appear.

The profit for a keyword depends on its cost, the number of searches which are expected to contain that keyword, and on the **click-through rate** for that keyword.

We have a limit on our advertising budget.

Application: Keyword advertising

Suppose we are a clothing retail website and we want to advertise our products via Google searches. When Google users search for terms such as “rain jackets”, we would like our ad to appear.

The profit for a keyword depends on its cost, the number of searches which are expected to contain that keyword, and on the **click-through rate** for that keyword.

We have a limit on our advertising budget.

(This is the static version of the problem, where we just choose a set of keywords to buy – no hyperheuristics for bidding.)

Application: Keyword advertising

- $n = 10$ keywords
- budget $W = 269$

Word	Profit	Cost
1	55	95
2	10	4
3	47	60
4	5	32
5	4	23
6	50	72
7	8	80
8	61	62
9	85	65
10	87	46

- Data from `f1_l-d_kp_10_269` in Bb.
- There are n relevant keywords. Keyword i has an associated cost w_i and expected profit v_i.
- We have to choose which set of keywords to buy, to maximise our expected profit, but we must not exceed our advertising budget W.

Knapsack problem

- $n = 10$ keywords
- budget $W = 269$

Item	Value	Weight
1	55	95
2	10	4
3	47	60
4	5	32
5	4	23
6	50	72
7	8	80
8	61	62
9	85	65
10	87	46

- This is a **Knapsack problem**: we must choose a subset of items from a set.
- Each item has a **weight** w_i and a **value** v_i.
- We can represent a solution by a set of items S.
- Our goal is to maximise the summed value of our subset $\sum_{i \in S} v_i$, without exceeding a constraint on weights $\sum_{i \in S} w_i \leq W$.

Tangent: can we solve this with IP?

Yes: define DVs x_i = whether to include item i in the set S.

The objective and constraints are linear.

But for large problems it becomes infeasible (recall that IP is much less scalable than LP), and a metaheuristic or constructive heuristic is needed.

A simple deterministic heuristic

- n = 10 keywords
- budget W = 269

Item	Value	Weight
10	87	46
9	85	65
8	61	62
1	55	95
6	50	72
3	47	60
2	10	4
7	8	80
4	5	32
5	4	23

Sort by decreasing value and take items one by one if they do not exceed weight limit.

A simple deterministic heuristic

- n = 10 keywords
- budget W = 269

Item	Value	Weight
10	87	46
9	85	65
8	61	62
1	55	95
6	50	72
3	47	60
2	10	4
7	8	80
4	5	32
5	4	23

Sort by decreasing value and take items one by one if they do not exceed weight limit.

- Choose **10**: val = 87; weight = 46; rem = 223

A simple deterministic heuristic

- n = 10 keywords
- budget W = 269

Item	Value	Weight
10	87	46
9	85	65
8	61	62
1	55	95
6	50	72
3	47	60
2	10	4
7	8	80
4	5	32
5	4	23

Sort by decreasing value and take items one by one if they do not exceed weight limit.

- Choose **10**: val = 87; weight = 46; rem = 223
- Choose **9**: val = 172; weight = 111; rem = 158

A simple deterministic heuristic

- n = 10 keywords
- budget W = 269

Item	Value	Weight
10	87	46
9	85	65
8	61	62
1	55	95
6	50	72
3	47	60
2	10	4
7	8	80
4	5	32
5	4	23

Sort by decreasing value and take items one by one if they do not exceed weight limit.

- Choose **10**: val = 87; weight = 46; rem = 223
- Choose **9**: val = 172; weight = 111; rem = 158
- Choose **8**: val = 233; weight = 173; rem = 96

A simple deterministic heuristic

- n = 10 keywords
- budget W = 269

Item	Value	Weight
10	87	46
9	85	65
8	61	62
1	55	95
6	50	72
3	47	60
2	10	4
7	8	80
4	5	32
5	4	23

Sort by decreasing value and take items one by one if they do not exceed weight limit.

- Choose **10**: val = 87; weight = 46; rem = 223
- Choose **9**: val = 172; weight = 111; rem = 158
- Choose **8**: val = 233; weight = 173; rem = 96
- Choose **1**: val = 288; weight = 268; rem = 1

A simple deterministic heuristic

- n = 10 keywords
- budget W = 269

Item	Value	Weight
10	87	46
9	85	65
8	61	62
1	55	95
6	50	72
3	47	60
2	10	4
7	8	80
4	5	32
5	4	23

Sort by decreasing value and take items one by one if they do not exceed weight limit.

- Choose **10**: val = 87; weight = 46; rem = 223
- Choose **9**: val = 172; weight = 111; rem = 158
- Choose **8**: val = 233; weight = 173; rem = 96
- Choose **1**: val = 288; weight = 268; rem = 1
- Cannot choose **6** (would exceed W)

A simple deterministic heuristic

- n = 10 keywords
- budget W = 269

Item	Value	Weight
10	87	46
9	85	65
8	61	62
1	55	95
6	50	72
3	47	60
2	10	4
7	8	80
4	5	32
5	4	23

Sort by decreasing value and take items one by one if they do not exceed weight limit.

- Choose **10**: val = 87; weight = 46; rem = 223
- Choose **9**: val = 172; weight = 111; rem = 158
- Choose **8**: val = 233; weight = 173; rem = 96
- Choose **1**: val = 288; weight = 268; rem = 1
- Cannot choose **6** (would exceed W)
- Cannot choose 3, 2, 7, 4, 5

A simple deterministic heuristic

- $n = 10$ keywords
- budget $W = 269$

Item	Value	Weight
10	87	46
9	85	65
8	61	62
1	55	95
6	50	72
3	47	60
2	10	4
7	8	80
4	5	32
5	4	23

Sort by decreasing value and take items one by one if they do not exceed weight limit.

- Choose **10**: val = 87; weight = 46; rem = 223
- Choose **9**: val = 172; weight = 111; rem = 158
- Choose **8**: val = 233; weight = 173; rem = 96
- Choose **1**: val = 288; weight = 268; rem = 1
- Cannot choose **6** (would exceed W)
- Cannot choose 3, 2, 7, 4, 5
- Solution $S = \{10, 9, 8, 1\}$ with value 288, weight 268.

Could randomised constructive heuristics help?

Recall that for TSP we considered using a randomised heuristic instead of a deterministic one.

A greedy randomised constructive heuristic for the Knapsack problem

1. Start with an empty set S
2. Find the unused items which would not exceed our weight limit
3. If there are none, finish
4. Choose the p highest-value items to form the **restricted candidates list** (RCL)
5. Choose one of these at random and add it to S
6. Go to 2.

A GRCH for the Knapsack problem

- n = 10 keywords
- budget W = 269

Item	Value	Weight
10	87	46
9	85	65
8	61	62
1	55	95
6	50	72
3	47	60
2	10	4
7	8	80
4	5	32
5	4	23

- RCL = 5, 4, 7, 2, 3, 6, 1, [8, 9, 10]

A GRCH for the Knapsack problem

- n = 10 keywords
- budget W = 269

Item	Value	Weight
10	87	46
9	85	65
8	61	62
1	55	95
6	50	72
3	47	60
2	10	4
7	8	80
4	5	32
5	4	23

- RCL = 5, 4, 7, 2, 3, 6, 1, [8, 9, 10]
- Eg choose **10**: weight = 46; rem = 223

A GRCH for the Knapsack problem

- n = 10 keywords
- budget W = 269

Item	Value	Weight
10	87	46
9	85	65
8	61	62
1	55	95
6	50	72
3	47	60
2	10	4
7	8	80
4	5	32
5	4	23

- RCL = 5, 4, 7, 2, 3, 6, 1, [8, 9, 10]
- Eg choose **10**: weight = 46; rem = 223
- RCL = 5, 4, 7, 2, 3, 6, [1, 8, 9]

A GRCH for the Knapsack problem

- n = 10 keywords
- budget W = 269

Item	Value	Weight
10	87	46
9	85	65
8	61	62
1	55	95
6	50	72
3	47	60
2	10	4
7	8	80
4	5	32
5	4	23

- RCL = 5, 4, 7, 2, 3, 6, 1, [8, 9, 10]
- Eg choose **10**: weight = 46; rem = 223
- RCL = 5, 4, 7, 2, 3, 6, [1, 8, 9]
- Eg choose **1**: weight = 141; rem = 128

A GRCH for the Knapsack problem

- n = 10 keywords
- budget W = 269

Item	Value	Weight
10	87	46
9	85	65
8	61	62
1	55	95
6	50	72
3	47	60
2	10	4
7	8	80
4	5	32
5	4	23

- RCL = 5, 4, 7, 2, 3, 6, 1, [8, 9, 10]
- Eg choose **10**: weight = 46; rem = 223
- RCL = 5, 4, 7, 2, 3, 6, [1, 8, 9]
- Eg choose **1**: weight = 141; rem = 128
- RCL = 5, 4, 7, 2, 3, [6, 8, 9]

A GRCH for the Knapsack problem

- n = 10 keywords
- budget W = 269

Item	Value	Weight
10	87	46
9	85	65
8	61	62
1	55	95
6	50	72
3	47	60
2	10	4
7	8	80
4	5	32
5	4	23

- RCL = 5, 4, 7, 2, 3, 6, 1, [8, 9, 10]
- Eg choose **10**: weight = 46; rem = 223
- RCL = 5, 4, 7, 2, 3, 6, [1, 8, 9]
- Eg choose **1**: weight = 141; rem = 128
- RCL = 5, 4, 7, 2, 3, [6, 8, 9]
- Eg choose **8**: weight = 203; rem = 66

A GRCH for the Knapsack problem

- n = 10 keywords
- budget W = 269

Item	Value	Weight
10	87	46
9	85	65
8	61	62
1	55	95
6	50	72
3	47	60
2	10	4
7	8	80
4	5	32
5	4	23

- RCL = 5, 4, 7, 2, 3, 6, 1, [8, 9, 10]
- Eg choose **10**: weight = 46; rem = 223
- RCL = 5, 4, 7, 2, 3, 6, [1, 8, 9]
- Eg choose **1**: weight = 141; rem = 128
- RCL = 5, 4, 7, 2, 3, [6, 8, 9]
- Eg choose **8**: weight = 203; rem = 66
- RCL = 5, 4, [2, 3, 9]

A GRCH for the Knapsack problem

- n = 10 keywords
- budget W = 269

Item	Value	Weight
10	87	46
9	85	65
8	61	62
1	55	95
6	50	72
3	47	60
2	10	4
7	8	80
4	5	32
5	4	23

- RCL = 5, 4, 7, 2, 3, 6, 1, [8, 9, 10]
- Eg choose **10**: weight = 46; rem = 223
- RCL = 5, 4, 7, 2, 3, 6, [1, 8, 9]
- Eg choose **1**: weight = 141; rem = 128
- RCL = 5, 4, 7, 2, 3, [6, 8, 9]
- Eg choose **8**: weight = 203; rem = 66
- RCL = 5, 4, [2, 3, 9]
- Eg choose **3**: weight = 263; rem = 6

A GRCH for the Knapsack problem

- n = 10 keywords
- budget W = 269

Item	Value	Weight
10	87	46
9	85	65
8	61	62
1	55	95
6	50	72
3	47	60
2	10	4
7	8	80
4	5	32
5	4	23

- RCL = 5, 4, 7, 2, 3, 6, 1, [8, 9, 10]
- Eg choose **10**: weight = 46; rem = 223
- RCL = 5, 4, 7, 2, 3, 6, [1, 8, 9]
- Eg choose **1**: weight = 141; rem = 128
- RCL = 5, 4, 7, 2, 3, [6, 8, 9]
- Eg choose **8**: weight = 203; rem = 66
- RCL = 5, 4, [2, 3, 9]
- Eg choose **3**: weight = 263; rem = 6
- RCL = [2]

A GRCH for the Knapsack problem

- n = 10 keywords
- budget W = 269

Item	Value	Weight
10	87	46
9	85	65
8	61	62
1	55	95
6	50	72
3	47	60
2	10	4
7	8	80
4	5	32
5	4	23

- RCL = 5, 4, 7, 2, 3, 6, 1, [8, 9, 10]
- Eg choose **10**: weight = 46; rem = 223
- RCL = 5, 4, 7, 2, 3, 6, [1, 8, 9]
- Eg choose **1**: weight = 141; rem = 128
- RCL = 5, 4, 7, 2, 3, [6, 8, 9]
- Eg choose **8**: weight = 203; rem = 66
- RCL = 5, 4, [2, 3, 9]
- Eg choose **3**: weight = 263; rem = 6
- RCL = [2]
- Choose **2**: weight = 267; rem = 2

A GRCH for the Knapsack problem

- n = 10 keywords
- budget W = 269

Item	Value	Weight
10	87	46
9	85	65
8	61	62
1	55	95
6	50	72
3	47	60
2	10	4
7	8	80
4	5	32
5	4	23

- RCL = 5, 4, 7, 2, 3, 6, 1, [8, 9, 10]
- Eg choose **10**: weight = 46; rem = 223
- RCL = 5, 4, 7, 2, 3, 6, [1, 8, 9]
- Eg choose **1**: weight = 141; rem = 128
- RCL = 5, 4, 7, 2, 3, [6, 8, 9]
- Eg choose **8**: weight = 203; rem = 66
- RCL = 5, 4, [2, 3, 9]
- Eg choose **3**: weight = 263; rem = 6
- RCL = [2]
- Choose **2**: weight = 267; rem = 2
- RCL empty: stop

A GRCH for the Knapsack problem

- n = 10 keywords
- budget W = 269

Item	Value	Weight
10	87	46
9	85	65
8	61	62
1	55	95
6	50	72
3	47	60
2	10	4
7	8	80
4	5	32
5	4	23

- RCL = 5, 4, 7, 2, 3, 6, 1, [8, 9, 10]
- Eg choose **10**: weight = 46; rem = 223
- RCL = 5, 4, 7, 2, 3, 6, [1, 8, 9]
- Eg choose **1**: weight = 141; rem = 128
- RCL = 5, 4, 7, 2, 3, [6, 8, 9]
- Eg choose **8**: weight = 203; rem = 66
- RCL = 5, 4, [2, 3, 9]
- Eg choose **3**: weight = 263; rem = 6
- RCL = [2]
- Choose **2**: weight = 267; rem = 2
- RCL empty: stop
- Solution $S = \{10, 1, 8, 3, 2\}$, weight 267, value 260

Other heuristics for the Knapsack problem

Above we took the **3 highest-value** items in the RCL.

- Another approach might define the **efficiency** of each item:

$$e_i = v_i / w_i$$

- We can also use Genetic Programming hyperheuristics to create good heuristics. See e.g. Drake et al. and Burke et al., Week 08 readings.

Generalising GRCH

The below code is **generic**, not problem-specific.

NB now minimising costs, not maximising values

```
def GRCH(
    instance,       # problem-specific data
    init,           # make an empty solution
    complete,       # test if a WIP sol is complete
    allowed_parts, # find remaining allowed parts
    cost,           # cost of adding a specified part
    add_part,       # add specified part to WIP
    p):             # prop. of eligible parts in RCL
```

GRCH

```
sol = init(instance)
while not complete(sol, instance):
    parts = allowed_parts(sol, instance)
    if len(parts) == 0: break # we must be finished
    costs = [cost(sol, part, instance)
             for part in parts]
    RCL_limit = np.percentile(costs, p * 100)
    # restricted candidates list
    RCL = [part for part, cost in zip(parts, costs)
           if cost <= RCL_limit]
    assert len(RCL)
    sol = add_part(sol, random.choice(RCL), instance)
return sol
```

GRCH for Knapsack

Knapsack instance:

```
instance = N, W, D
```

where `N` is number of items, `W` is weight limit, and `D` (shape `(N, 2)`) contains the values and weights.

We can then **plug in** problem-specific functions:

GRCH for Knapsack

```
def knapsack_init(instance): return set()

def knapsack_complete(sol, instance):
    # if all items added, or weight equals limit
    N, W, D = instance
    w = sum(D[i][1] for i in sol)
    return (len(sol) == N) or (w >= W)
```

GRCH for Knapsack

```
def knapsack_allowed_parts(sol, instance):
    # unused item allowed if its weight would fit
    N, W, D = instance
    w = sum(D[i][1] for i in sol)
    return [i for i, item in enumerate(D) if
            i not in sol and item[1] <= W - w]
```

GRCH for Knapsack

```
def knapsack_cost(sol, item, instance):
    N, W, D = instance
    # [0] is value, so invert with "-"
    return -D[item][0]

def knapsack_add_part(sol, item, instance):
    sol.add(item)
    return sol
```

GRCH for Knapsack

And we call it like this:

```
knapsack_instance = N, W, D # read from a file

sol = GRCH(knapsack_instance,
           knapsack_init,
           knapsack_complete,
           knapsack_allowed_parts,
           knapsack_cost,
           knapsack_add_part,
           p)
```

GRCH for Knapsack

And we call it like this:

```
knapsack_instance = N, W, D # read from a file

sol = GRCH(knapsack_instance,
           knapsack_init,
           knapsack_complete,
           knapsack_allowed_parts,
           knapsack_cost,
           knapsack_add_part,
           p)
```

If we want to solve a different problem such as TSP, we keep GRCH and just plug-in data and functions suitable for TSP.

Overview

- Opening the black box (a little)
- A constructive heuristic for knapsack problems
- **GRASP**

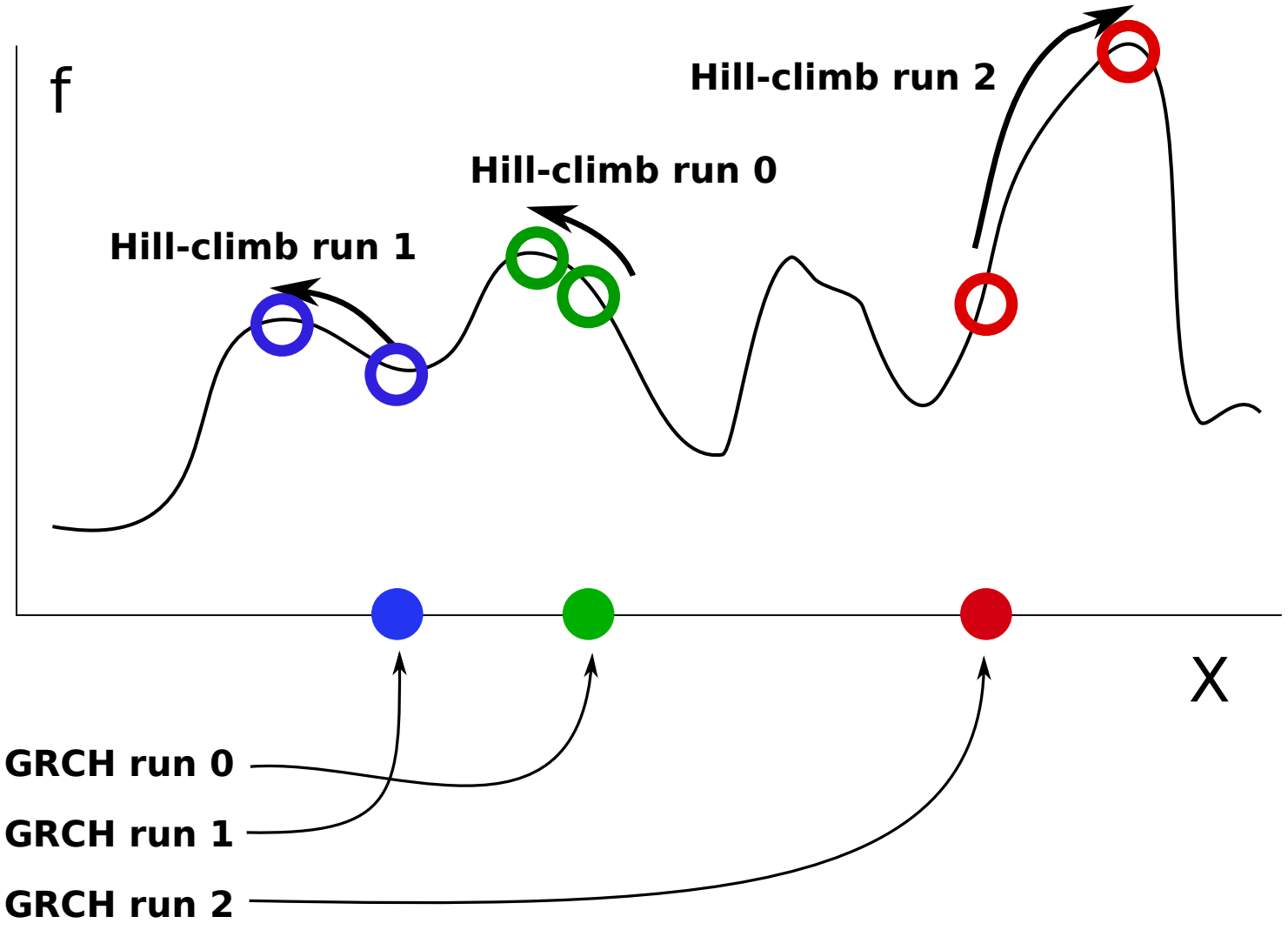
Greedy Randomised Adaptive Search Procedure (GRASP)

GRASP is a **hybrid** of a constructive heuristic and a metaheuristic. This is the simple version:

1. x = GRCH()
2. Use hill-climbing starting at x until it stops improving
3. If our fitness evaluation budget is not exhausted, go to 1.

GRASP is due to Feo and Resende (1989). Some more sophisticated versions also exist. See Luke, **Essentials**, Chapter 8.

GRASP


f
Hill-climb run 2
Hill-climb run 0
Hill-climb run 1
X
GRCH run 0
GRCH run 1
GRCH run 2

Why the name?

- Recall: a **greedy** algorithm is one that makes the best choice it can right now, ignoring the future
- GRASP is greedy because at each step in the constructive heuristic, it focusses on the best items, not looking to the future
- It is randomised because it doesn't always pick the **very best** item
- It is **adaptive** because it uses hill-climbing search.

Non-deterministic

- The advantage of using a non-deterministic heuristic in this context: we get **multiple** good, feasible starting points for multiple hill-climbs
- Hopefully some in good **basins of attraction**
- So, GRASP is like iterated hill-climbing with **good, feasible** starting points.

Exam timetabling

The problem of **exam timetabling** is a real-world problem and there are other similar **scheduling** problems in industry.

- Multiple time-slots (some discouraged, e.g. weekend)
- Multiple rooms (each with a capacity)
- Multiple exams (each with a class list and instructor)

Constraints:

- Each exam scheduled exactly once
- For each room, each time-slot, sum of exam class sizes does not exceed room capacity
- Avoid student taking two exams at once
- Many more! https://www.unitime.org/exam_description.php

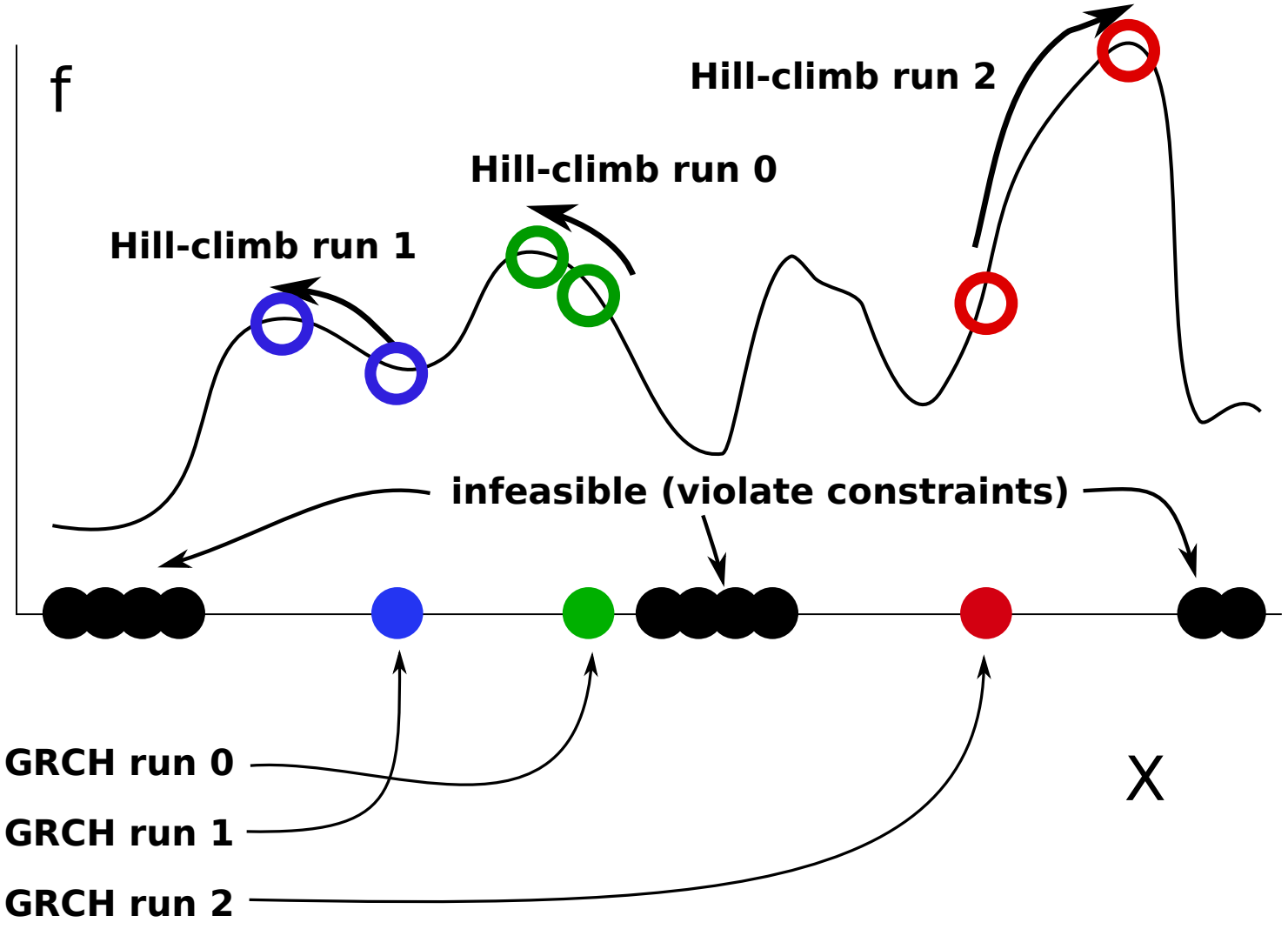
Exam timetabling

- Key point: finding a **feasible** solution is hard, so hard to write `init()` for (e.g.) hill-climbing.

Exam timetabling

- Key point: finding a **feasible** solution is hard, so hard to write `init()` for (e.g.) hill-climbing.
- Idea: use GRCH as an `init()` function for (iterated) hill-climbing.

GRASP


f
Hill-climb run 2
Hill-climb run 0
Hill-climb run 1
infeasible (violate constraints)
GRCH run 0
GRCH run 1
GRCH run 2
X

More hybrid algorithms

(Hybrids of constructive heuristics and metaheuristics)

- Juan et al.: instead of choosing uniformly from the RCL, choose from a **triangular distribution**

More hybrid algorithms

(Hybrids of constructive heuristics and metaheuristics)

- Juan et al.: instead of choosing uniformly from the RCL, choose from a **triangular distribution**
- De Armas et al.: simulated annealing where the mutate function discards part of the solution and then uses a constructive heuristic to **repair it**

More hybrid algorithms

(Hybrids of constructive heuristics and metaheuristics)

- Juan et al.: instead of choosing uniformly from the RCL, choose from a **triangular distribution**
- De Armas et al.: simulated annealing where the mutate function discards part of the solution and then uses a constructive heuristic to **repair it**
- Moraglio and McDermott: a universal representation, the **program trace**, generalises this so we can plug in any metaheuristic `init()` instead of the constructive heuristic.

Recap

- Opening the black-box (a little)
- Gray box manifesto
- Problems:
  - **TSP** (e.g. circuit design)
  - **Knapsack** (e.g. static version of keyword bidding)
  - **Scheduling**: (e.g. exam timetabling: very difficult to produce a feasible solution)
- Algorithms:
  - **Constructive heuristics** versus metaheuristics
  - **Greedy** heuristics
  - **Deterministic** versus **non-deterministic**
  - **GRCH** is a generic version
  - **Hybrids** (e.g. **GRASP** uses GRCH and a local search metaheuristic).

Further reading (optional)

- Norvig's great intro to TSP: http://nbviewer.jupyter.org/url/norvig.com/ipython/TSP.ipynb
- TSPLIB benchmark data: http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html
- U Waterloo TSP page: http://www.math.uwaterloo.ca/tsp/
- All of Norvig's PyTudes are great (mostly not related to optimisation): https://github.com/norvig/pytudes

Overview

- Opening the black box (a little)
- A constructive heuristic for knapsack problems
- GRASP