Multi-Objective Optimisation - Part 2

November 26, 2024

1 Multi Objective Evolutionary Algorithms (Part 2)

In this chapter we continue with multi-objective search algorithms, and consider some alternatives to NSGA-II. First we need to import the usual stuff, set up our wrapper class and plotting infrastructure.

1.1 Setting the Scene

[1]:
```
import sys
import random
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
from math import sqrt
```

As an example problem instance, we consider the Next Release Problem (NRP) again.

[2]:
```
profit_map = {}         # customer id => weight
requirements_map = {}   # customer id => [ requirement_id * ]
cost_map = {}           # requirement id => cost
dependency_map = {}     # requirement id => [ requirement_id * ]
num_requirements = 140
```

We'll need our parser for the standard format again.

[3]:
```
def parse_nrp(filename):

    with open(filename) as fp:
        levels   = int(fp.readline())
        req_id = 1
        for level in range(levels):
            num_reqs = int(fp.readline())
            for cost in fp.readline().split():
                cost_map[req_id] = int(cost)
                req_id += 1
        total_deps = int(fp.readline())
        for num_dep in range(total_deps):
            r1, r2 = fp.readline().split()
            r1_id = int(r1)
            if r1_id in dependency_map:
                dependency_map[int(r1)].append(int(r2))
            else:
                dependency_map[int(r1)] = [int(r2)]

        total_customers = int(fp.readline())
        for num_cust in range(total_customers):
            customer = fp.readline().split()
            profit_map[num_cust + 1] = int(customer[0])
            num_reqs = int(customer[1])
            requirements = []
            for num_req in range(num_reqs):
                requirements.append(int(customer[2+num_req]))
            requirements_map[num_cust + 1] = requirements

        num_requirements = len(cost_map)
```

As running example, we will use the small instance `nrp1.txt`:

[4]:
```
parse_nrp("data/nrp/nrp1.txt")
```

Our first objective is to maximise the profit:

[5]:
```
# Profit is the sum of the weights of customers whose requirements are satisfied
def function1(solution):
    fitness = 0

    requirements = set([x+1 for x in range(len(solution)) if solution[x] == 1])
    for customer_id in profit_map.keys():
        reqs = set(requirements_map[customer_id])
        if reqs.issubset(requirements):
            fitness += profit_map[customer_id]

    return fitness
```

The second objective is to minimise the costs:

[6]:
```
# Cost is the sum of costs of the implemented requirements
def function2(solution):
    cost = 0

    for i in range(num_requirements):
        if solution[i]:
            cost += cost_map[i+1]

    return cost
```

[7]:
```
total_costs = sum([cost_map[i+1] for i in range(num_requirements)])
```

[8]:
```python
total_profit = sum(profit_map.values())
```

To avoid redundantly calculating fitness values, we will use the wrapper class for the list to cache values in attributes.

[9]:
```python
class L(list):
    """
    A subclass of list that can accept additional attributes.
    Should be able to be used just like a regular list.
    """
    def __new__(self, *args, **kwargs):
        return super(L, self).__new__(self, args, kwargs)

    def __init__(self, *args, **kwargs):
        if len(args) == 1 and hasattr(args[0], '__iter__'):
            list.__init__(self, args[0])
        else:
            list.__init__(self, args)
        self.__dict__.update(kwargs)

    def __call__(self, **kwargs):
        self.__dict__.update(kwargs)
        return self
```

We can now store the fitness values of individuals as attributes of the objects

[10]:
```python
def evaluate(individual):
    individual.fitness1 = function1(individual)
    individual.fitness2 = function2(individual)
```

Individuals of NRP solutions are instances of `L` rather than lists, and we can define our usual search operators for bitvector representation:

[11]:
```python
def get_random_individual():
    individual = L(random.choice([0,1]) for _ in range(num_requirements))
    evaluate(individual)
    return individual
```

[12]:
```python
def mutate(solution):
    P_mutate = 1/len(solution)
    mutated = L(solution[:])
    for position in range(len(solution)):
        if random.random() < P_mutate:
            mutated[position] = 1 - mutated[position]
    evaluate(mutated)
    return mutated
```

[13]:
```python
def crossover(parent1, parent2):
    pos = random.randint(0, len(parent1))
    offspring1 = L(parent1[:pos] + parent2[pos:])
    offspring2 = L(parent2[:pos] + parent1[pos:])
    return (offspring1, offspring2)
```

In addition to the representation, we also want to keep using animations to study how the algorithms explore the search space:

[14]:
```
ims = []  # global variable to store images of the animation

def initialise_plot():
    global ims
    global fig
    global ax

    ims = []

    %matplotlib agg
    fig, ax = plt.subplots()
    plt.xlabel('Profit', fontsize=15)
    plt.ylabel('Cost', fontsize=15)
    ims = []
    %matplotlib inline
```

For each iteration of the algorithm, we will update this animation with a snapshot of the current population and their fitness values:

[15]:
```
def plot(population):
    function1_values = [x.fitness1 for x in population]
    function2_values = [x.fitness2 for x in population]

    ims.append((ax.scatter(function1_values, function2_values, color="blue"),))
```

1.2 Baseline 1: Random Search

When studying different search algorithms, random search always servers as a sanity check to compare against. To generalise random search to multi-objective random search, we'll need to define the dominance relation again.

[16]:
```
def dominates(solution1, solution2):
    """
     A solution x(1) is said to dominate the other solution x(2) if both␣
 ↪condition 1 and 2 below are true:

     Condition 1: x(1) is no worse than x(2) for all objectives
     Condition 2: x(1) is strictly better than x(2) in at least one objective

     We are maximising fitness 1, but minimising fitness 2
    """
    if solution1.fitness1 < solution2.fitness1 or solution1.fitness2 >␣
↪solution2.fitness2:
        return False

    if solution1.fitness1 > solution2.fitness1 or solution1.fitness2 <␣
↪solution2.fitness2:
        return True

    return False
```

An operation we will frequently need in this chapter, and also for random search, is to extract the non-dominated solutions for a given collection of solutions.

[17]:
```
def get_nondominated(population):
    nondominated = []

    for x in population:
        dominated = False
        for y in population:
            if dominates(y, x):
                dominated = True
                break
        if not dominated:
            nondominated.append(x)

    return nondominated
```

Given these ingredients, we can generalise random search to multi-objective random search by repeatedly sampling random individuals, and keeping those that are not dominated.

[18]:
```
max_gen = 100
population_size = 20
```

[19]:
```
def random_moo():
    initialise_plot()
    result = []
    for iteration in range(max_gen * population_size):
        candidate = get_random_individual()
        result = get_nondominated(result + [candidate])
        if iteration % population_size == 0:
            plot(result)
    return result
```

Similar to the last chapter, we can look at the evolution for each algorithm by plotting the solutions found for each iteration (or in this case, after each `population_size` individuals have been evaluated, to speed up the animation).

[20]:
```
result = random_moo()
```

[21]:
```
im_ani = animation.ArtistAnimation(fig, ims, interval=50, repeat_delay=3000,␣
 ↪blit=True)
HTML(im_ani.to_jshtml())
```

[21]:
```
<IPython.core.display.HTML object>
```

To systematically compare sarch algorithms, we need to run experiments with multiple repetitions again. For this we use our usual helper function.

[22]:
```
from IPython.utils import io

def run_times(algorithm, repetitions):
    global ims
    result = []
    for i in range(repetitions):
        ims = []
        with io.capture_output() as captured:
            front = algorithm()
        result.append(front)
    return result
```

We also need a metric to compare algorithms with. In the last chapter we considered several different metrics, in particular the hypervolume.

[23]:
```
def hypervolume(front, r):
    front.sort(key=lambda i: i.fitness1)

    hv = (abs(r[0] - front[0].fitness1) / total_profit) * (abs(r[1] - front[0].
 ↪fitness2) / total_costs)

    for i in range(1, len(front)):
        hv += (abs(front[i-1].fitness1 - front[i].fitness1) / total_profit) *␣
 ↪(abs(r[1] - front[i].fitness2) / total_costs)

    return hv
```

We will keep all experiment results in a dictionary `results` such that we can produce comparative boxplots throughout this chapter.

[24]:
```
fronts_random = run_times(random_moo, 10)
results = {}
results["Random"] = [hypervolume(front, (0,total_costs)) for front in␣
 ↪fronts_random]
```

1.3 Baseline 2: NSGA-II

In the last chapter we introduced NSGA-II as our first multi-objective search algorithm, and indeed it is one of the most popular multi-objective algorithms and has been shown to be effective and efficient for many different problems. Hence we definitely need to include it in our comparison in this chapter. In the following, the individual bits and pieces of NSGA-II from the previous chapter are re-introduced.

[25]:
```python
def fast_non_dominated_sort(solutions):
    front = [[]]

    S = [[] for _ in range(len(solutions))]
    n = [0 for _ in range(len(solutions))]

    for p in range(len(solutions)):
        S[p] = []
        n[p] = 0
        for q in range(len(solutions)):
            if dominates(solutions[p], solutions[q]):
                S[p].append(q)
            elif dominates(solutions[q], solutions[p]):
                n[p] = n[p] + 1

        if n[p] == 0:
            front[0].append(p)
            solutions[p].rank = 0

    i = 0
    while front[i]:
        Q = []
        for p in front[i]:
            for q in S[p]:
                n[q] = n[q] - 1
                if n[q] == 0:
                    Q.append(q)
                    solutions[q].rank = i + 1
        i = i + 1
        front.append(Q)

    del front[-1]
    return front
```

[26]:
```python
def calculate_crowding_distance_and_sort(front):

    data = [(x, front[x].fitness1, front[x].fitness2) for x in␣
 ↪range(len(front))]
    sorted1 = [(x, y) for (x, y, z) in sorted(data, key=lambda tup: tup[1])]
    sorted2 = [(x, z) for (x, y, z) in sorted(data, key=lambda tup: tup[2])]

    distance = [0 for _ in range(0,len(front))]
    range_fitness1 = max(x.fitness1 for x in front) - min(x.fitness1 for x in␣
 ↪front)
```

```
    range_fitness2 = max(x.fitness2 for x in front) - min(x.fitness2 for x in␣
↪front)

    distance[sorted1[0][0]] = sys.maxsize
    distance[sorted1[-1][0]] = sys.maxsize

    distance[sorted2[0][0]] = sys.maxsize
    distance[sorted2[-1][0]] = sys.maxsize

    for k in range(1,len(front)-1):
        index = sorted1[k][0]
        distance[index] = distance[index] + (sorted1[k+1][1] - sorted1[k-1][1])␣
↪/ range_fitness1

    for k in range(1,len(front)-1):
        index = sorted2[k][0]
        distance[index] = distance[index] + (sorted2[k+1][1] - sorted2[k-1][1])␣
↪/ range_fitness2

    for k in range(0, len(front)):
        front[k].distance = distance[k]

    front.sort(key = lambda i: i.distance, reverse=True)
```

[27]:
```
def binary_rank_tournament(population):
    individual1 = random.choice(population)
    individual2 = random.choice(population)

    if individual1.rank < individual2.rank:
        return individual1
    elif individual1.rank > individual2.rank:
        return individual2
    else:
        return max([individual1, individual2], key = lambda i: i.distance)
```

[28]:
```
def generate_offspring(population):
    offspring_population = []
    while len(offspring_population) < len(population):
        parent1 = binary_rank_tournament(population)
        parent2 = binary_rank_tournament(population)
        if random.random() < P_xover:
            offspring1, offspring2 = crossover(parent1, parent2)
        else:
            offspring1, offspring2 = parent1, parent2

        offspring1 = mutate(offspring1)
        offspring2 = mutate(offspring2)
```

```python
        offspring_population.append(offspring1)
        offspring_population.append(offspring2)

    return offspring_population
```

[29]:
```python
def get_initial_population(population_size):
    population = [get_random_individual() for _ in range(population_size)]
    fronts = fast_non_dominated_sort(population)

    for front_indices in fronts:
        front = [population[index] for index in front_indices]
        calculate_crowding_distance_and_sort(front)

    return population
```

[30]:
```python
def nsgaii():
    initialise_plot()
    population = get_initial_population(population_size)
    offspring_population = generate_offspring(population)

    for iteration in range(max_gen):
        combined = population + offspring_population
        #plot(combined)
        fronts = fast_non_dominated_sort(combined)
        population = []

        for front_indices in fronts:
            front = [combined[index] for index in front_indices]
            calculate_crowding_distance_and_sort(front)

            for i in front:
                population.append(i)
                if len(population) == population_size:
                    break
            if len(population) == population_size:
                break

        plot(population)
        offspring_population = generate_offspring(population)

    non_dominated_sorted_solution = fast_non_dominated_sort(population)
    result = [population[x] for x in non_dominated_sorted_solution[0]]
    plot(result)

    return result
```

First let's consider the evolution of an individual run again.

```
[31]: P_xover = 0.7
      result = nsgaii()
```

```
[32]: im_ani = animation.ArtistAnimation(fig, ims, interval=50, repeat_delay=3000,␣
       ↪blit=True)
      HTML(im_ani.to_jshtml())
```

```
[32]: <IPython.core.display.HTML object>
```

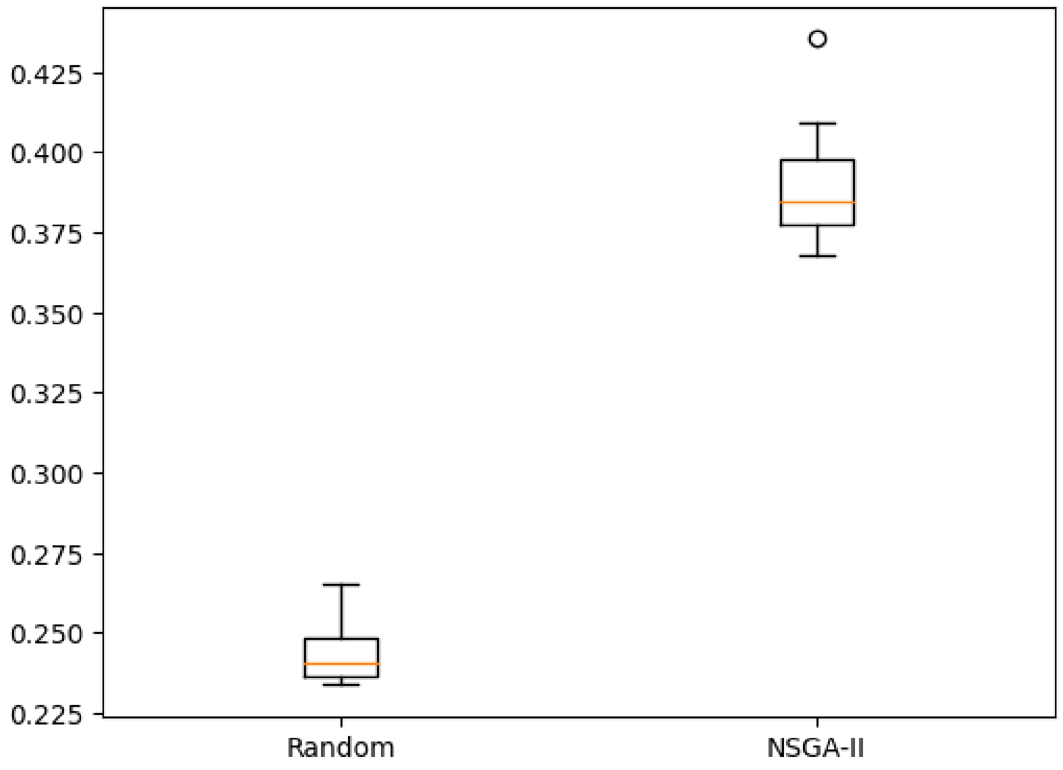
We also need to collect some more data for comparison.

```
[33]: fronts_nsgaii = run_times(nsgaii, 10)
      results["NSGA-II"] = [hypervolume(front, (0,total_costs)) for front in␣
       ↪fronts_nsgaii]
```

Our first sanity check is whether NSGA-II is indeed better than random search.

```
[34]: fig, ax = plt.subplots()
      ax.boxplot(results.values())
      ax.set_xticklabels(results.keys())
```

```
[34]: [Text(1, 0, 'Random'), Text(2, 0, 'NSGA-II')]
```



1.4 PAES

As first alternative example of a multi-objective search algorithm, we consider the Pareto Archived Evolution Strategy (PAES):

J. Knowles, D. Corne. "The pareto archived evolution strategy: A new baseline algorithm for pareto multiobjective optimisation." In Congress on Evolutionary Computation (CEC99) (Vol. 1, pp. 98-105). 1999

PAES is intended to be the simplest possible non-trivial algorithm capable of generating diverse solutions in the Pareto optimal set. It essentiall is a (1+1)ES adapted for multi-objective search, and is intended as baseline approach for evaluation of other algorithms.

Like a (1+1)ES the algorithm has a population of size 1 and produces 1 offspring. In addition, there is an archive which stores non-dominated solutions (subject to diversity criteria). At the end of the run, the archive is the final result.

1. Generate random current solution
2. Evaluate and add to archive
3. While not done:
4. Create candidate by mutating current solution
5. If candidate is dominated by current, reject it
6. Else if current is dominated by the candidate, accept candidate and add it to the archive
7. Else Compare candidate solutions with archive members, update archive and current

The archive stores only non-dominated solutions, and has a maximum size. The archive is also used in order to decide whether or not to accept an offspring: - Candidates which dominate the archive are always accepted and archived. - Candidates which are dominated by the archive are always rejected. - Non-dominated are accepted/archived based on how many similar individuals already exist

When an individual is added to the archive, we need to remove all individuals from the archive that are dominated by this new member. However, when the maximum archive size is reached, we may have to decide which individuals to keep in the archive. In this case, PAES checks if the new candidate would increase the diversity in the archive. If so, it replaces the individual risiding in the most crowded part of the archive.

We start by defining some helper functions since PAES works with an archive. First, we add a function that tells us if a candidate individual is dominated by any of the individuals in the archive:

[35]:
```python
def is_dominated(candidate, archive):
    for i in archive:
        if dominates(i, candidate):
            return True
    return False
```

We also need an update operation that removes all dominated individuals from an archive:

[36]:
```python
def remove_dominated(candidate, archive):
    copy = [x for x in archive if not dominates(candidate, x)]
    archive.clear()
    archive.extend(copy)
```

Third,we need to select all solutions in the archive that are dominated by an individual, as we want to remove all of them when adding a new non-dominated solution:

[37]:
```python
def get_dominated(candidate, archive):
    return [x for x in archive if dominates(candidate, x)]
```

Finally, we need an operation that updates the archive. As we are using lists but technically the archive should be an set for PAES, we'll have to add a check here (inefficient, but shorter than redefining previous operations to work on sets rather than lists):

[38]:
```python
def add_to_archive(individual, archive):
    if not individual in archive:
        archive.append(individual)
```

When there is dominance, the choice of what to do is easy. The more tricky case is when the new individual is not dominated by the archive. As the archive has a maximum size, we need to decide which individuals to keep once we hit the maximum. In PAES, this is done using a grid from which we can infer which areas of the objective space are more crowded.

[39]:
```python
grid_size = 10

def get_grid(archive):
    grid = {}
    max_profit = max([i.fitness1 for i in archive])+1
    max_effort = max([i.fitness2 for i in archive])+1
    f1step = max_profit / grid_size
    f2step = max_effort / grid_size

    for x in range(grid_size):
        grid[x] = {}
        for y in range(grid_size):
            grid[x][y] = []

    for individual in archive:
        x = individual.fitness1 // f1step
        y = individual.fitness2 // f2step
        grid[x][y].append(individual)

    return grid
```

We want to add individuals to the grid if they help us preserve diversity. Thus, we check if we would add the individual to the most crowded grid cell. If we are, then we are not increasing diversity.

[40]:
```python
def increases_diversity(candidate, grid, archive):
    max_crowd = 0
    target_crowd = 0

    # We have to redundantly calculate this to figure out the step size...
    max_profit = max([i.fitness1 for i in archive]) + 1
    max_cost = max([i.fitness2 for i in archive]) + 1
    f1step = max_profit / grid_size
    f2step = max_cost / grid_size

    target_x = candidate.fitness1 // f1step
    target_y = candidate.fitness2 // f2step

    for x in range(grid_size):
        for y in range(grid_size):
            num = len(grid[x][y])
            if num > max_crowd:
                max_crowd = num
            if target_x == x and target_y == y:
                target_crowd = num

    return target_crowd < max_crowd
```

We can also compare individuals in terms of how crowded their grid cells are:

[41]:
```
def less_crowded_than(individual1, individual2, grid):
    individual1_crowd = 0
    individual2_crowd = 0

    for x in range(grid_size):
        for y in range(grid_size):
            if individual1 in grid[x][y]:
                individual1_crowd = len(grid[x][y])
            if individual2 in grid[x][y]:
                individual2_crowd = len(grid[x][y])

    return individual1_crowd < individual2_crowd
```

If we need to reduce the size of the archive, we randomly pick one individual from the most crowded grid cell:

[42]:
```
def remove_from_grid(grid, archive):
    pos_x, pos_y = 0, 0
    max_crowd = 0

    for x in range(grid_size):
        for y in range(grid_size):
            num = len(grid[x][y])
            if num > max_crowd:
                max_crowd = num
                pos_x, pos_y = x, y

    selected = random.choice(grid[pos_x][pos_y])
    archive.remove(selected)
```

The PAES algorithm itself mainly consists of the logic to decide what to do in case of non-domination:

[43]:
```python
def paes():
    initialise_plot()
    archive = []
    current = get_random_individual()
    add_to_archive(current, archive)

    for step in range(population_size * max_gen):
        candidate = mutate(current)
        while candidate == current or candidate in archive:
            candidate = mutate(current)
        if step % population_size == 0:
            plot(archive + [candidate])

        if not dominates(current, candidate) and not is_dominated(candidate,␣
↪archive):
            dominated_archive = get_dominated(candidate, archive)
            if dominated_archive:
                # If the candidate dominates something in the archive
                # we keep it in the archive and make it the new current
                remove_dominated(candidate, archive)
                add_to_archive(candidate, archive)
                current = candidate
            else:
                if len(archive) == population_size:
                    # Maximum archive size reached
                    grid = get_grid(archive)
                    if increases_diversity(candidate, grid, archive):
                        remove_from_grid(grid, archive)
                        add_to_archive(candidate, archive)
                        grid = get_grid(archive)
                        if less_crowded_than(candidate, current, grid):
                            current = candidate
                else:
                    # Enough space in archive
                    add_to_archive(candidate, archive)
                    grid = get_grid(archive)
                    if less_crowded_than(candidate, current, grid):
                        current = candidate

    return get_nondominated(archive)
```

[44]:
```python
result = paes()
```

```
[45]: im_ani = animation.ArtistAnimation(fig, ims, interval=50, repeat_delay=3000,␣
        ↪blit=True)
      HTML(im_ani.to_jshtml())
```

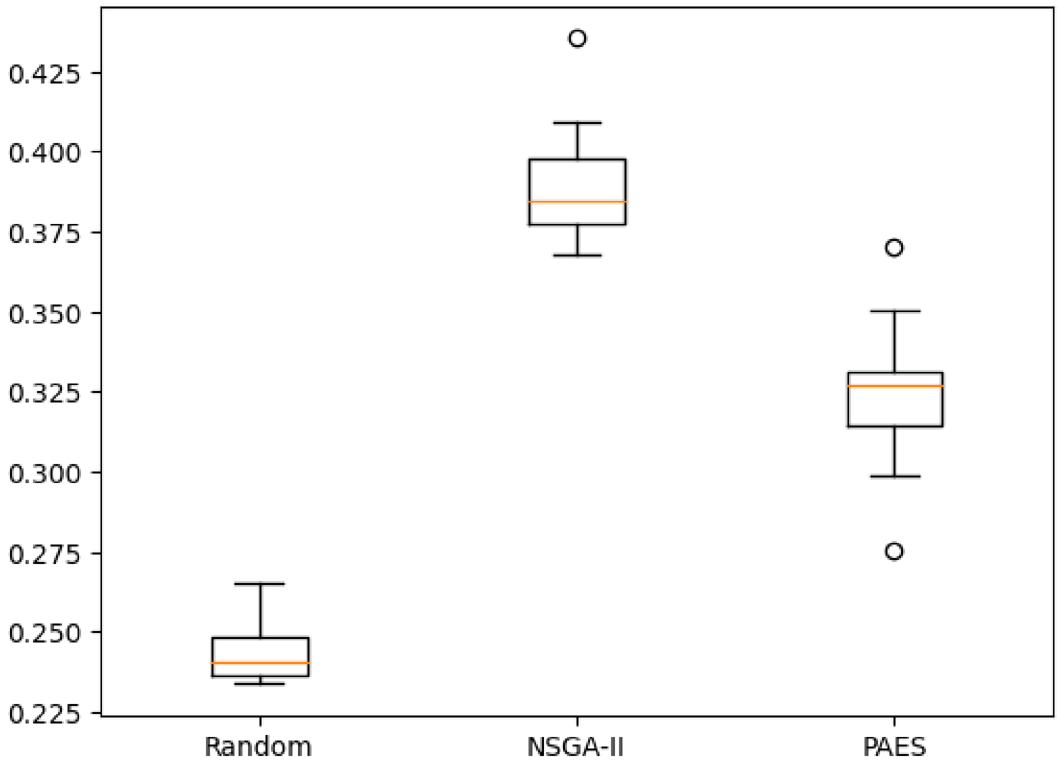
```
[45]: <IPython.core.display.HTML object>
```

As always, we add some data and compare.

```
[46]: fronts_paes = run_times(paes, 10)
      results["PAES"] = [hypervolume(front, (0,total_costs)) for front in fronts_paes]
```

```
[47]: fig, ax = plt.subplots()
      ax.boxplot(results.values())
      ax.set_xticklabels(results.keys())
```

```
[47]: [Text(1, 0, 'Random'), Text(2, 0, 'NSGA-II'), Text(3, 0, 'PAES')]
```



1.5 SPEA-2

The Strength Pareto Evolutionary Algorithm (SPEA) version 2 is often treated as the main competitor of NSGA-II. It was defined in the following paper:

E. Zitzler, M. Laumanns and L. Thiele, "SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization", in Evolutionary Methods for Design, Optimisation and

Control with Application to Industrial Problems, 2002.

This algorithm assigns a fitness to an individual based on the strength of its dominators. This fitness value is then minimised to produce solutions that are not dominated.

Another central distinguishing feature is the use of an archive.

The overall workflow of the algorithm is as follows: 1. Create initial population P_0 and create an empty archive $\bar{P}_0$. 2. Evaluate fitness values of the individuals in P_t and $\bar{P}_t$ 3. Copy all non-dominated individuals in P_t and $\bar{P}_t$ to $\bar{P}_{t+1}$. If $\bar{P}_{t+1}$ exceeds the maximum archive size then truncate; if it is smaller then fill up with dominated individuals from P_t and $\bar{P}_t$. 4. Perform binary tournament selection with replacement on $\bar{P}_{t+1}$ to fill the mating pool 5. Apply recombination and mutation operators to the mating pool and set P_{t+1} to the resulting population. 6. Repeat from step 2 until done

The solution is represented by the non-dominated individuals in $\bar{P}$ at the end.

Let's first look at how the fitness values are determined. The score of an individual is the number of solutions that it dominates:

$Score(i) = |j \mid j \in P_t \cup \bar{P}_t \wedge i \succ j|$

Here, $i \succ j$ denotes that i dominates j. (Note that Zitzler et al. use the operator $\succ$ the other way round as Deb et al.)

[48]:
```
def score(individual, combined):
    score = 0
    for other in combined:
        if dominates(individual, other):
            score += 1

    individual.score = score
```

The raw fitness value of an individual is calculated as the sum of its dominators' strengths:

[49]:
```
def raw_fitness(individual, combined):
    raw_fitness = 0
    for other in combined:
        if dominates(other, individual):
            raw_fitness += other.score

    individual.raw_fitness = raw_fitness
```

The fitness value considers not only the raw fitness, but also the density, such that individuals in less populated areas of the search space are preferred:

$Density(i) = \frac{1}{\sigma_i^k + 2}$

Here, σ_i^k denotes the distance of i to its kth neighbour, and $k = \sqrt{N + \bar{N}}$, where N is the size of the population and $\bar{N}$ is the size of the archive.

The distance σ is defined in terms of the objective values:

[50]:
```python
def get_distance(individual, other):
    sum = 0

    sum += (individual.fitness1 - other.fitness1) ** 2
    sum += (individual.fitness2 - other.fitness2) ** 2

    return sqrt(sum)
```

Now we just need to calculate these distances, sort, and assign to the individuals:

[51]:
```python
def density(individual, combined):
    k = int(sqrt(len(combined)))

    distances = []
    for j in range(len(combined)):
        other = combined[j]
        if individual == other:
            continue
        distances.append(1/(2.0 + get_distance(individual, other)))
    distances.sort()
    individual.distance = distances[k]
```

To put it all together, the following function calculates the fitness (strength) of all individuals given a population and archive:

[52]:
```python
def calculate_strength(population, archive):
    combined = population + archive

    for x in combined:
        score(x, combined)
    for x in combined:
        raw_fitness(x, combined)
    for x in combined:
        density(x, combined)

    for individual in combined:
        individual.fitness = individual.raw_fitness + individual.distance
```

The other unique aspect of this algorithm is the handling of the archive. The size of the archive is constant over time, which means that sometimes individuals need to be removed or added in order to adjust the size.

The *truncate* function removes individuals from the archive; at each step, the individual which has the minimum distance to another individual is chosen for removal until the archive is no longer too large. When multiple individuals have the same distance to their closest neighbour, they are compared against the next closest neighbour until a difference is found. For this, we need a custom comparison operator:

[53]:
```python
def compare_distances(distances1, distances2):
    for i in range(len(distances1)):
        if distances1[i] != distances2[i]:
            return distances1[i] - distances2[i]
    return 0
```

The truncate function now needs to determine all distances and then sort the individuals in order to decide which ones to drop:

[54]:
```python
from functools import cmp_to_key
def truncate(archive):
    # Remove individual with minimum distance
    num_remove = abs(population_size - len(archive))

    for i in range(len(archive)):
        individual = archive[i]
        distances = []
        for j in range(len(archive)):
            if i == j:
                continue
            other = archive[j]
            distances.append(get_distance(individual, other))
        distances.sort()
        individual.distances = distances

        # This ignores the case that multiple individuals have the same distance
        # in which case we need to look at the next distance

    def compare(x, y):
        return compare_distances(x.distances, y.distances)
    archive.sort(key=cmp_to_key(compare))
    for i in range(num_remove):
        del archive[0]
```

Since Python >= 3 only supports the use of keys to sort lists, this function uses some extra code to wrap out `compare_distances` function as a key function.

If the archive is too small, then we fill it with the best dominated individuals from the population. Thus, we sort the population by fitness (i.e., strength), and then pick from the sorted list.

[55]:
```python
def pad_archive(next_archive, population, archive):
    num_missing = population_size - len(next_archive)

    # fill with dominated individuals in archive and population
    candidates = [i for i in population+archive if i.fitness >= 1]
    candidates.sort(key = lambda r : r.fitness)
    next_archive.extend(candidates[:num_missing])
```

The mating is the same we already know from other evolutionary algorithms. The selection operator used is a binary tournament similar to what NSGA-II uses. However, unlike the binary tournament in NSGA-II, the tournament is now decided by the strength (fitness) function:

[56]:
```
def binary_tournament_fitness(population):
    individual1 = random.choice(population)
    individual2 = random.choice(population)

    if individual1.fitness < individual2.fitness:
        return individual1
    elif individual1.fitness > individual2.fitness:
        return individual2
    else:
        return random.choice([individual1, individual2])
```

Selected individuals are subjected to crossover and mutation as usual:

[57]:
```
def generate_offspring_spea2(population):
    offspring_population = []
    while len(offspring_population) < population_size:
        parent1, parent2 = binary_tournament_fitness(population),␣
 ↪binary_tournament_fitness(population)
        if random.random() < P_xover:
            offspring1, offspring2 = crossover(parent1, parent2)
        else:
            offspring1, offspring2 = parent1, parent2
        offspring1 = mutate(offspring1)
        offspring2 = mutate(offspring2)

        offspring_population.append(offspring1)
        offspring_population.append(offspring2)

    return offspring_population
```

Now we have all the components in place and can implement the overall algorithm:

[58]:
```
def spea2():
    initialise_plot()
    population = [get_random_individual() for _ in range(population_size)]
    archive = []

    for iteration in range(max_gen):
        plot(population+archive)

        calculate_strength(population, archive)

        # Copy all non-dominated individuals in population and archive to next␣
 ↪archive
        next_archive = []
        for i in population + archive:
            if i.fitness < 1:
                next_archive.append(i)

        if len(next_archive) < population_size:
            pad_archive(next_archive, population, archive)

        if len(next_archive) > population_size:
            truncate(next_archive)


        # Mating selection:
        population = generate_offspring_spea2(next_archive)
        archive = next_archive

        # Termination: return non-dominated individuals in next_archive
        result = [p for p in next_archive if p.fitness < 1]

    plot(result)

    return result
```

[59]:
```
result = spea2()
```

[60]:
```
im_ani = animation.ArtistAnimation(fig, ims, interval=50, repeat_delay=3000,␣
 ↪blit=True)
HTML(im_ani.to_jshtml())
```

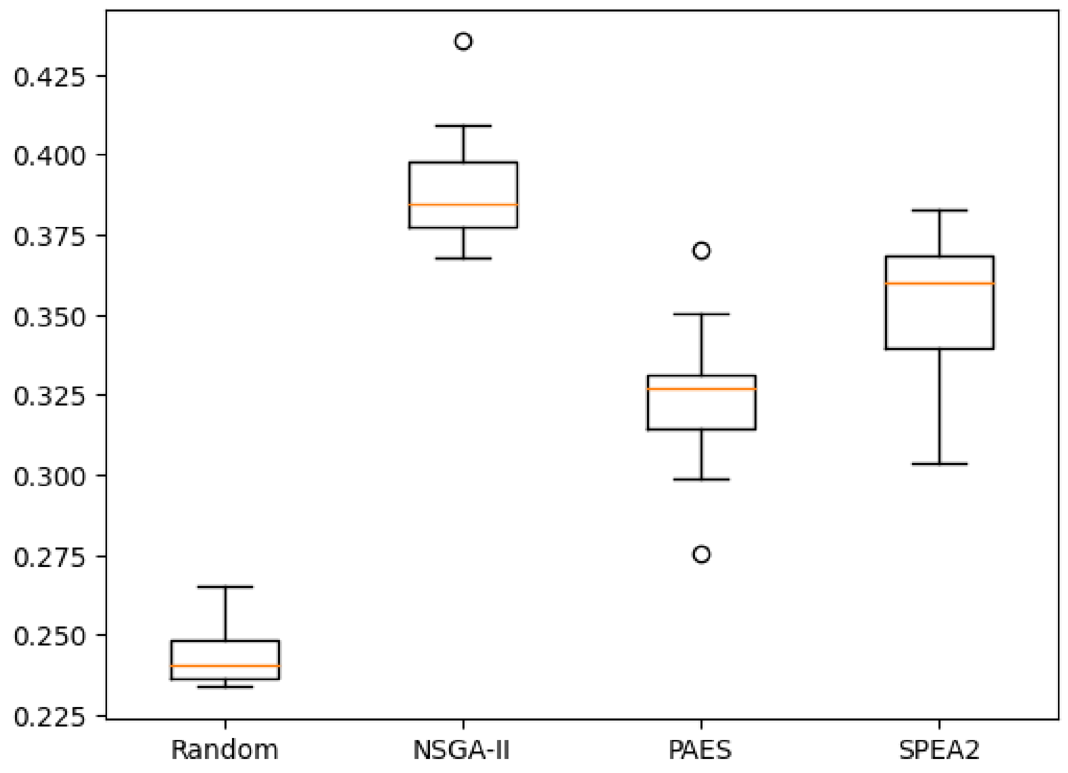
[60]:
```
<IPython.core.display.HTML object>
```

As usual we add some datapoints for this algorithm.

[61]:
```
fronts_spea2 = run_times(spea2, 10)
results["SPEA2"] = [hypervolume(front, (0,total_costs)) for front in␣
 ↪fronts_spea2]
```

[62]:
```
fig, ax = plt.subplots()
ax.boxplot(results.values())
ax.set_xticklabels(results.keys())
```

[62]:
```
[Text(1, 0, 'Random'),
 Text(2, 0, 'NSGA-II'),
 Text(3, 0, 'PAES'),
 Text(4, 0, 'SPEA2')]
```



1.6 Two Archives

The Two Archives algorithm generalises the idea of using archives and uses one for convergence, and another one for diversity:

- If a new solution is not dominated by both archives and dominates at least one solution in either archive, it goes into the convergence archive.
- If a new solution is not dominated by both archives but fails to dominate any solution in either archive, it goes into the diversity archive.
- When the archives get full, the convergence archive is preserved while the diversity archive is pruned based on crowding distance.

Praditwong, K., & Yao, X. (2006, November). A new multi-objective evolutionary optimisation algorithm: The two-archive algorithm. In 2006 International Conference on Computational Intelligence and Security (Vol. 1, pp. 286-291). IEEE.

The general workflow is as follows: 1. Initialise the population (as usual) 2. Initialise both archives as empty sets 3. Evaluate the initial population 4. Repeat until done: 5. Collect non-dominated individuals to the archives 6. Generate new population with parents from archives 7. Evaluate new population

During selection, one of the archives is probabilistically chosen. The probability is a pre-defined parameter that is a ratio to choose members from the convergence archive to the diversity archive. A member in the chosen archive is selected uniformly at random.

[63]:
```
def twoarchive_selection(convergence_archive, diversity_archive):
    if not convergence_archive:
        return random.choice(diversity_archive)
    if not diversity_archive:
        return random.choice(convergence_archive)
    if random.random() < 0.5:
        return random.choice(convergence_archive)
    else:
        return random.choice(diversity_archive)
```

The selection is the only difference in terms of reproduction compared to other evolutionary algorithms we have seen previously:

[64]:
```
def twoarchive_generate_offspring(convergence_archive, diversity_archive):
    offspring_population = []
    while len(offspring_population) < population_size:
        parent1 = twoarchive_selection(convergence_archive, diversity_archive)
        parent2 = twoarchive_selection(convergence_archive, diversity_archive)
        if random.random() < P_xover:
            offspring1, offspring2 = crossover(parent1, parent2)
        else:
            offspring1, offspring2 = parent1, parent2
        offspring1 = mutate(offspring1)
        offspring2 = mutate(offspring2)

        offspring_population.append(offspring1)
        offspring_population.append(offspring2)

    return offspring_population
```

We need to select all individuals from the resulting population that are not dominated by the archives. For this, we need some helper functions. First, we define a helper function that tells us whether a candidate solution is dominated by an archive:

A new individual is first compared with all members in the current archives. If it is dominated by a member of the archives, it is discarded, otherwise it becomes a new member of the archives.

The remainder of the archives are compared with the new member and two cases are possible: - If the new member dominates a member of the archives then the dominated member is removed and the new member is received by the *convergence* archive - If the new member does not dominate any members and is not dominated by any archive members then it becomes a member of the diversity archive, and the size of the diversity archive is increased.

[65]:
```
def collect_nondominated(individual, ca, da):
    # TODO: During this stage, any duplicated member is deleted.

    if is_dominated(individual, ca) or is_dominated(individual, da):
        return
    dominated_ca = get_dominated(individual, ca)
    dominated_da = get_dominated(individual, da)
    if len(dominated_ca) + len(dominated_da) > 0:
        for i in dominated_ca:
            ca.remove(i)
        for i in dominated_da:
            da.remove(i)
        ca.append(individual)
    else:
        # No individuals are dominated
        da.append(individual)
```

If the total size of the archives overflows, then we need to remove individuals. The removal operator deletes only members in the diversity archive and has no impact on the convergence archive.

To select which members to remove from the diversity archive, we calculate the shortest distance to the nearest member in the convergence archive for all members in the diversity archive. The member with the shortest distance among the diversity members is deleted until the total size equals the capacity.

[66]:
```
archive_size = 40

def remove(ca, da):
    if len(ca) + len(da) > archive_size:
        for individual in da:
            individual.length = sys.maxsize
            for other in ca:
                d = get_distance(individual, other)
                if individual.length > d:
                    individual.length = d
        da.sort(key=lambda r: r.length)
        while len(ca) + len(da) > archive_size:
            del da[0]
```

Now we just need to put everything together.

[67]:
```
def twoarchives():
    initialise_plot()
    population = [get_random_individual() for _ in range(population_size)]
    convergence_archive = []
    diversity_archive   = []

    for iteration in range(max_gen):
        plot(population+convergence_archive+diversity_archive)

        # Collect non-dominated individuals
        for i in get_nondominated(population):
            collect_nondominated(i, convergence_archive, diversity_archive)
        # Truncate archive sizes
        remove(convergence_archive, diversity_archive)

        next_generation = twoarchive_generate_offspring(convergence_archive,␣
↪diversity_archive)
        population = next_generation

    result = get_nondominated(convergence_archive)
    plot(result)
    return result
```

[68]:
```
result = twoarchives()
```

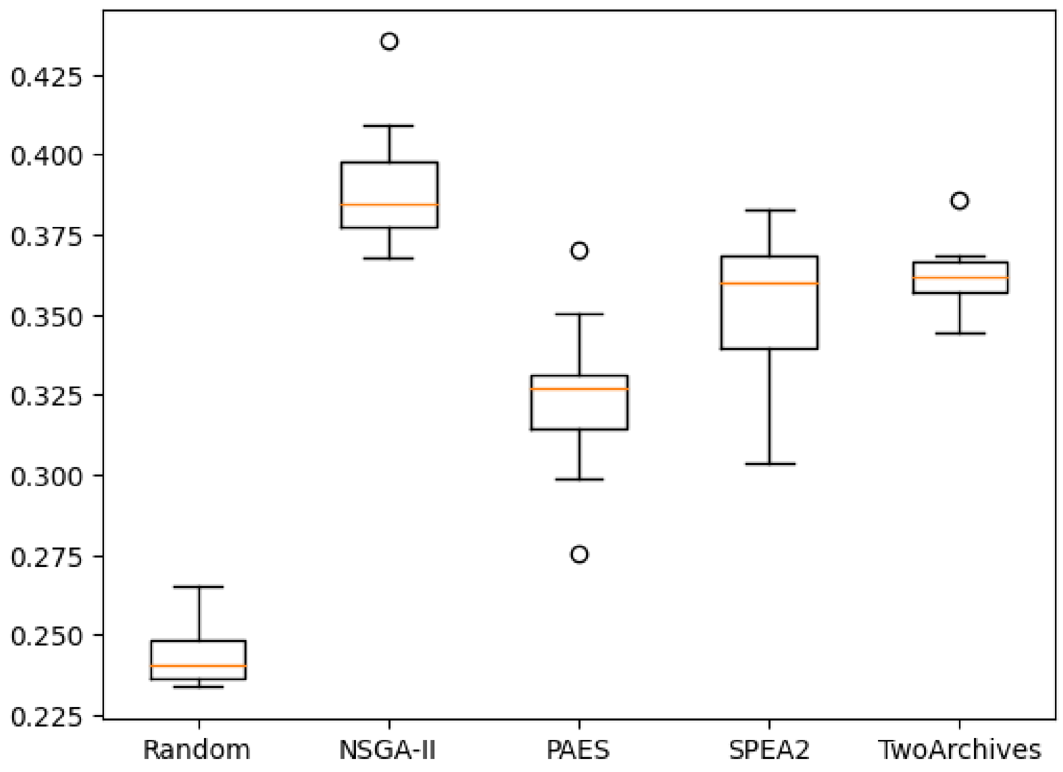
[69]:
```
im_ani = animation.ArtistAnimation(fig, ims, interval=50, repeat_delay=3000,␣
↪blit=True)
HTML(im_ani.to_jshtml())
```

[69]:
```
<IPython.core.display.HTML object>
```

[70]:
```
fronts_ta = run_times(twoarchives, 10)
results["TwoArchives"] = [hypervolume(front, (0,total_costs)) for front in␣
↪fronts_ta]
```

[71]:
```
fig, ax = plt.subplots()
ax.boxplot(results.values())
ax.set_xticklabels(results.keys())
```

[71]:
```
[Text(1, 0, 'Random'),
 Text(2, 0, 'NSGA-II'),
 Text(3, 0, 'PAES'),
 Text(4, 0, 'SPEA2'),
 Text(5, 0, 'TwoArchives')]
```



1.7 SMS-EMOA

We've been comparing the multi-objective algorithms in terms of the hypervolume, although they used domination and diversity measurements to guide the search. An alternative is to use the hypervolume *directly* to drive the search. The hypervolume is often referred to as S metric, and the idea to optimise the hypervolume is implemented in the S metric selection evolutionary multi objective algorithm (SMS-EMOA):

Beume, N., Naujoks, B., & Emmerich, M. (2007). SMS-EMOA: Multiobjective selection based on dominated hypervolume. European Journal of Operational Research, 181(3), 1653-1669.

The overall principle of SMS-EMOA is the following: 1. Generate a random population of individuals 2. Repeat until done: 3. Generate an offspring by variation 4. Replace existing member of population if it improves the hypervolume

The *reduce* function sorts the population using fast non-dominated sort (as introduced with NSGA-II), and then discards one individual from the worst front. The choice of which individual to discard is determined by the contribution the hypervolume: The individual contributing least to the hypervolume is the one discarded.

[72]:
```python
def reduce(population):
    fronts = fast_non_dominated_sort(population)
    last_front = [population[index] for index in fronts[-1]]
    if len(last_front) > 1:
        hypervolumes = []
        for i in range(len(last_front)):
            front = last_front[:i] + last_front[i+1:]
            hypervolumes.append(hypervolume(front, (0,total_costs)))
        chosen = last_front[hypervolumes.index(max(hypervolumes))]
        population.remove(chosen)
    else:
        population.remove(last_front[0])
```

Since the calculation of the hypervolume can be computationally expensive,the algorithm is implemented as a steady state algorithm.

[73]:
```
def sms():
    initialise_plot()
    population = [get_random_individual() for _ in range(population_size)]

    for step in range(max_gen * population_size):
        if step % population_size == 0:
            plot(population)

        parent1, parent2 = random.choice(population), random.choice(population)
        if random.random() < P_xover:
            offspring1, offspring2 = crossover(parent1, parent2)
        else:
            offspring1, offspring2 = parent1, parent2

        population.append(mutate(random.choice([offspring1, offspring2])))
        reduce(population)

    result = get_nondominated(population)
    plot(result)
    return result
```

[74]:
```
result = sms()
```

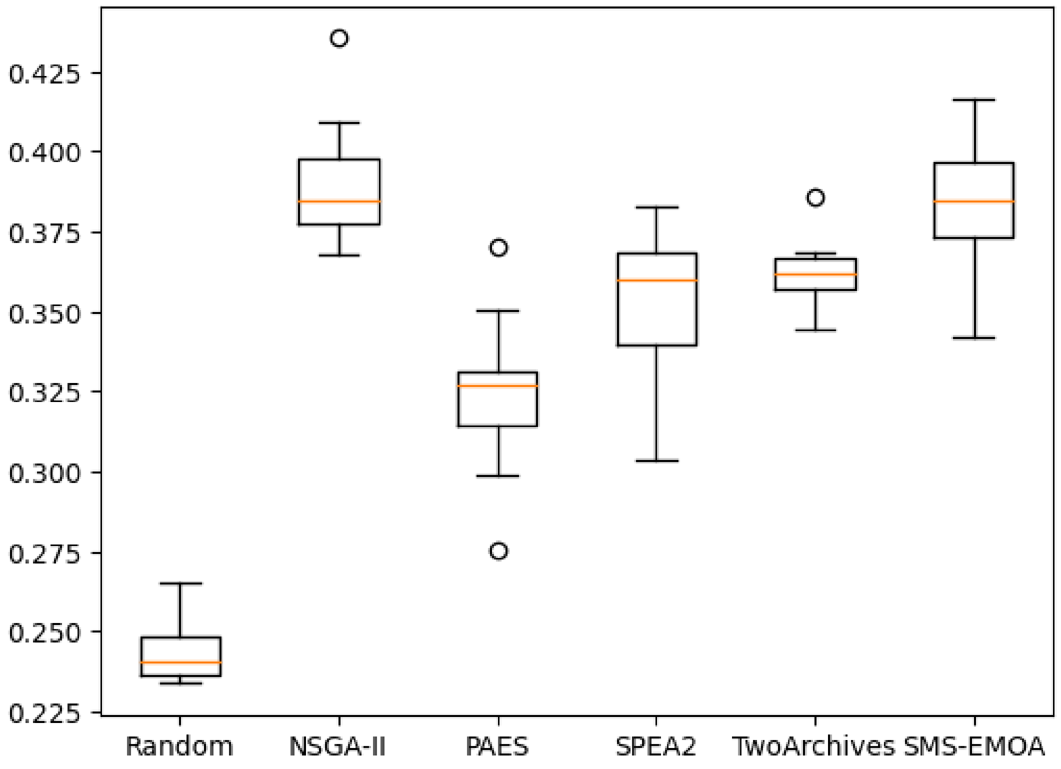
[75]:
```
im_ani = animation.ArtistAnimation(fig, ims, interval=50, repeat_delay=3000,␣
 ↪blit=True)
HTML(im_ani.to_jshtml())
```

[75]:
```
<IPython.core.display.HTML object>
```

[76]:
```
fronts_sms = run_times(sms, 10)
results["SMS-EMOA"] = [hypervolume(front, (0,total_costs)) for front in␣
 ↪fronts_sms]
```

[77]:
```
fig, ax = plt.subplots()
ax.boxplot(results.values())
ax.set_xticklabels(results.keys())
```

```
[77]: [Text(1, 0, 'Random'),
       Text(2, 0, 'NSGA-II'),
       Text(3, 0, 'PAES'),
       Text(4, 0, 'SPEA2'),
       Text(5, 0, 'TwoArchives'),
       Text(6, 0, 'SMS-EMOA')]
```



This tells us which algorithm is best overall on our first NRP problem.

1.8 Alternative Next Release Problem

```
[78]: profit_map = {}         # customer id => weight
      requirements_map = {}   # customer id => [ requirement_id * ]
      cost_map = {}           # requirement id => cost
      dependency_map = {}     # requirement id => [ requirement_id * ]
      num_requirements = 620
      parse_nrp("data/nrp/nrp2.txt")
      total_costs = sum([ cost_map[i+1] for i in range(num_requirements)])
      total_profit = sum(profit_map.values())
```

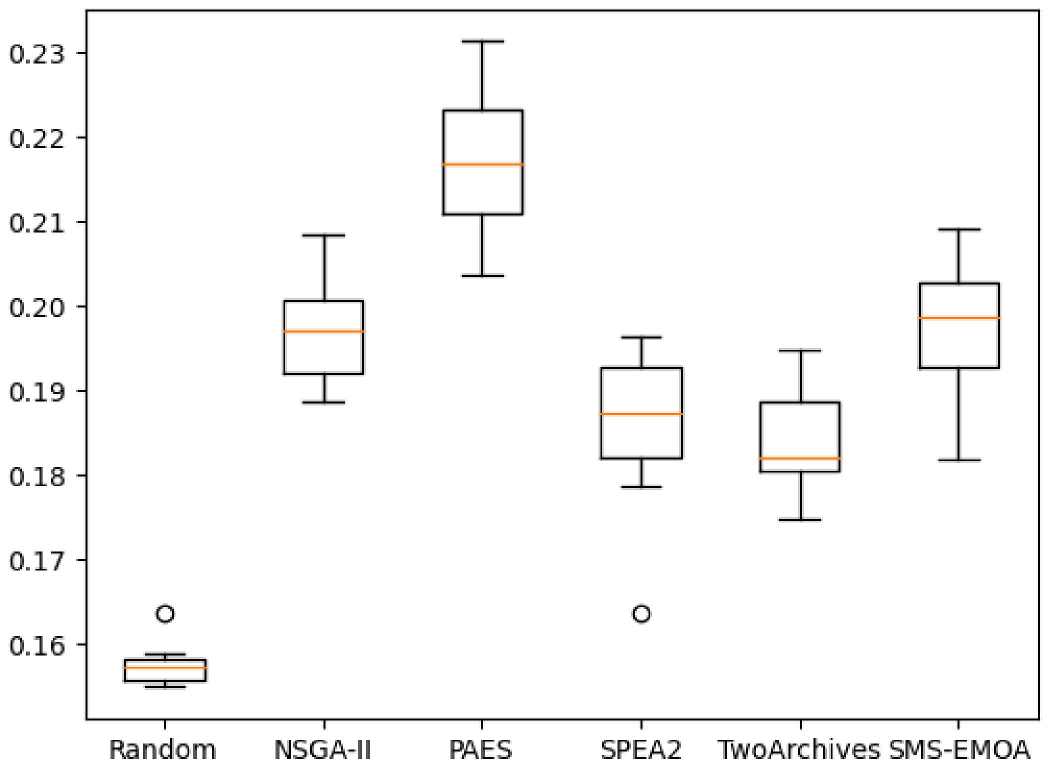
```
[79]: results = {}
```

```
[80]: fronts_random = run_times(random_moo, 10)
      fronts_nsgaii = run_times(nsgaii, 10)
      fronts_paes = run_times(paes, 10)
      fronts_spea2 = run_times(spea2, 10)
      fronts_ta = run_times(twoarchives, 10)
      fronts_sms = run_times(sms, 10)

      results["Random"]   = [hypervolume(front, (0,total_costs)) for front in␣
       ↪fronts_random]
      results["NSGA-II"]  = [hypervolume(front, (0,total_costs)) for front in␣
       ↪fronts_nsgaii]
      results["PAES"] = [hypervolume(front, (0,total_costs)) for front in fronts_paes]
      results["SPEA2"] = [hypervolume(front, (0,total_costs)) for front in␣
       ↪fronts_spea2]
      results["TwoArchives"] = [hypervolume(front, (0,total_costs)) for front in␣
       ↪fronts_ta]
      results["SMS-EMOA"] = [hypervolume(front, (0,total_costs)) for front in␣
       ↪fronts_sms]
```

```
[81]: fig, ax = plt.subplots()
      ax.boxplot(results.values())
      ax.set_xticklabels(results.keys())
```

```
[81]: [Text(1, 0, 'Random'),
       Text(2, 0, 'NSGA-II'),
       Text(3, 0, 'PAES'),
       Text(4, 0, 'SPEA2'),
       Text(5, 0, 'TwoArchives'),
       Text(6, 0, 'SMS-EMOA')]
```



1.9 Comparison

There is a range of benchmark problems to compare multi-objective optimisation algorithms on. For example, we will consider the ZDT set of functions defined in the following paper:

Chase, N., Rademacher, M., Goodman, E., Averill, R., & Sidhu, R. (2009). A benchmark study of multi-objective optimization methods. BMK-3021, Rev, 6, 1-24.

The ZDT1 function has a convex Pareto-optimal front. The objective functions are

$f_1(x) = x_1 \; f_2(x) = g(x)[1 - \sqrt{x_1/g(x)}]$

where $g(x)$ is defined as:

$g(x) = 1 + 9(\Sigma_{i=2}^{n} x_i)/(n-1)$.

Individuals are vectors of floating point numbers of size n:

[82]:
```
n = 30
```

[83]:
```
def function1(individual):
    return individual[0]
```

[84]:
```
def function2(individual):
    x0 = individual[0]
    sum = 0.0;
    for x in individual[1:]:
        sum += x

    g = 1.0 + 9.0 * sum / (len(individual) - 1)

    return g * (1 - sqrt(x0 / g))
```

Our representation currently only gives us bitvectors, so we need to adapt the search operators to produce vectors of floating point numbers.

[85]:
```
def get_random_individual():
    individual = L(random.random() for _ in range(n))
    evaluate(individual)
    return individual
```

To mutate a floating point number, we add some random noise using a Gaussian distribution.

[86]:
```
def mutate(solution):
    P_mutate = 1/len(solution)
    mutated = L(solution[:])
    for position in range(len(solution)):
        if random.random() < P_mutate:
            mutated[position] = max(0, min(1, mutated[position] + random.
 ↪gauss(0, 0.05)))
    evaluate(mutated)
    return mutated
```

Both our objectives are minimisation problems, so we need to update the dominance relation accordingly.

[87]:
```
def dominates(solution1, solution2):
    if solution1.fitness1 > solution2.fitness1 or solution1.fitness2 >␣
 ↪solution2.fitness2:
        return False

    if solution1.fitness1 < solution2.fitness1 or solution1.fitness2 <␣
 ↪solution2.fitness2:
        return True

    return False
```

We also need to remove the normalisation based on total profit and costs.

[88]:
```
max_fitness1 = function1([1 for _ in range(n)])
max_fitness2 = function2([1 for _ in range(n)])
```

[89]:
```
def hypervolume(front, r):
    front.sort(key=lambda i: i.fitness1)

    hv = (abs(r[0] - front[0].fitness1) / max_fitness1) * (abs(r[1] - front[0].
 ↪fitness2) / max_fitness2)

    for i in range(1, len(front)):
        hv += (abs(front[i-1].fitness1 - front[i].fitness1) / max_fitness1) *␣
 ↪(abs(r[1] - front[i].fitness2) / max_fitness2)

    return hv
```

Finally, we also need to fix the axis labels on our animation plots.

[90]:
```
ims = []  # global variable to store images of the animation

def initialise_plot():
    global ims
    global fig
    global ax

    ims = []

    %matplotlib agg
    fig, ax = plt.subplots()
    plt.xlabel('Function 1', fontsize=15)
    plt.ylabel('Function 2', fontsize=15)
    ims = []
    %matplotlib inline
```

[91]:
```
random_moo()
```

[91]:
```
[[0.0007792786681323216,
  0.28644039928701115,
  0.2340185984580273,
  0.2862896412451973,
  0.401177430732536,
  0.6397008115256076,
  0.40854017628066175,
  0.4571659979550352,
  0.6731476501821813,
  0.9181105706432097,
  0.5081114763700095,
  0.32910612112007576,
  0.2591577497126928,
  0.8379509684964153,
  0.1675973683702775,
  0.45917239917735386,
  0.12449203980706591,
  0.397689933073497,
  0.7773774793857988,
  0.613200963857503,
  0.33910706356130205,
  0.7340596128016225,
  0.2174372355712214,
  0.28568878510103535,
  0.7311768777843172,
  0.09282015998689919,
  0.2524177391805964,
  0.01854139089264417,
  0.6122905660938933,
  0.3236453390882399],
 [0.042603953172036135,
  0.23652067882675254,
  0.6751779734668901,
  0.6495952310768208,
  0.2491257864216212,
  0.14585227913953203,
  0.5760010902399859,
  0.4351664915952791,
  0.05310340476874176,
  0.1969440725967041,
  0.8456256728312285,
  0.007652712906543013,
  0.7561699515934835,
  0.023775673899302463,
  0.10727420091907636,
  0.0865136414553086,
  0.5248358621281115,
  0.2615404782112327,
  0.5824480378449683,
  0.12952235900110476,
  0.8727690077260752,
  0.22299657985413812,
  0.5664946626806788,
  0.33589348462091484,
  0.6627783645899815,
  0.05272464606425986,
  0.5062043481479698,
  0.5273065962699792,
  0.5193708802097041,
  0.42148335174837803],
 [0.06576131448250111,
  0.6024451997611167,
  0.27558372431221445,
  0.6017926102769692,
  0.13046963472698703,
  0.01908999192320404,
  0.20986483737045514,
  0.20030199375151247,
  0.919832403686037,
  0.3679619405393848,
  0.3680758740438569,
  0.9144044403415411,
  0.06663591048308337,
  0.5502644382946218,
  0.6252608598495245,
  0.8529488089312427,
  0.10117014472464658,
  0.4592943413377356,
  0.3367729433523087,
  0.09826310392237336,
  0.003173559105533852,
  0.09740401144425637,
  0.8857732253722129,
  0.15698281750744014,
  0.961071628314219,
  0.46667441358742123,
  0.6367182699711259,
  0.017919883885233356,
  0.056252740703254256,
  0.11457350579692449],
 [0.013190860418369255,
  0.1947405400592872,
  0.5681714316419088,
  0.3164077301343228,
  0.5432605480300412,
  0.3945540383247642,
  0.48281774430718605,
  0.23620264750596964,
  0.23018662871094842,
  0.7219687503580116,
  0.768541116567834,
  0.14615742487452588,
  0.6024471819535068,
  0.9851519932507012,
  0.6092325366833947,
  0.5163656334134449,
  0.26669460231079556,
  0.5927543317461463,
  0.37060191315115276,
  0.4041353704900733,
  0.31506102229717914,
  0.7628173335194199,
  0.11864125989029972,
  0.016067163699802767,
  0.9090642875821732,
  0.03108731609089721,
  0.7246044256759141,
  0.3756515806398305,
  0.05095965850569073,
  0.09214364729050184],
 [0.7373568936711683,
  0.7296307395153762,
  0.08400027035899083,
  0.9345007743831423,
  0.9966363117741548,
  0.10212713924619077,
  0.11616680495409959,
  0.0714696596683061,
  0.18867115268673196,
  0.8456300410115619,
  0.07760453669213285,
  0.23521896307804202,
  0.16611139280233322,
  0.9013872098686124,
  0.37830070941209515,
  0.2056230693106691,
  0.2510046044026455,
  0.5995453162208622,
  0.3447720173857135,
  0.29308322110006646,
  0.10453416444311381,
  0.046634288440428606,
  0.13130852696120865,
  0.3083820363177422,
  0.054804313421500384,
  0.07620794519707086,
  0.4958021597197775,
  0.5536892967086685,
  0.845690851837266,
  0.07121045716190189],
 [0.39427520544957717,
  0.05542226717005372,
  0.9570383856346412,
  0.1258224877930595,
  0.024391961208989565,
  0.37844341732207143,
  0.7504595534202936,
  0.03730315369338766,
  0.8238283655995802,
  0.026182226729250746,
  0.2747779810732245,
  0.5561902140224821,
  0.8241983991406817,
  0.13599779392574074,
  0.21925874544751922,
  0.23523070353951925,
  0.10557608253137651,
  0.09839760262870312,
  0.0019432176858010797,
  0.5535859402250848,
  0.06142264373854078,
  0.22347160462035198,
  0.03175240855153383,
  0.8341929051896172,
  0.2282119391861257,
  0.298850300598576,
  0.14726085931367616,
  0.34735660453839035,
  0.8438346467629487,
  0.11579016640671724],
 [0.22793333327156007,
  0.3487249168338651,
  0.5538747933846331,
  0.8479412925334912,
  0.7500765712580874,
  0.6038968957889848,
  0.08644927187945461,
  0.3320265488638885,
  0.08457281379972392,
  0.23602911836952412,
  0.1512034186124982,
  0.569451585787104,
  0.2976462387110429,
  0.11799240372017228,
  0.2699204690810857,
  0.1366105451463866,
  0.31929007904554474,
  0.027780654760547585,
  0.20866231715846095,
  0.120030120415487,
  0.19660999656567335,
  0.148213428615371,
  0.013153341846059097,
  0.3722648004255682,
  0.738366747984691,
  0.26033604154664014,
  0.619590843978461,
  0.7976234108412067,
  0.23206842040353604,
  0.547042286103131],
 [0.21319710615983922,
  0.4334323208150327,
  0.2666423024332101,
  0.12365504035457031,
  0.14007495449171048,
  0.37959661798868427,
  0.2377846399604987,
  0.08583305251566031,
  0.319172652557948,
  0.23111490451689143,
  0.31971307562124707,
  0.10900225294600419,
  0.08375004341060177,
  0.3340821951571703,
  0.3498075355615978,
  0.6435567277644694,
  0.6117170690192032,
  0.198650232812414,
  0.2826998206224365,
  0.06529595607738115,
  0.6265604122414389,
  0.9520159942297202,
  0.7121652242542107,
  0.2604230256154523,
  0.571373383460944,
  0.12204282192803628,
  0.7530246944727946,
  0.2949826403874861,
  0.04494189445644581,
  0.38898632519611454],
 [0.9119456061644403,
  0.5440964143689832,
  0.0371111758667555,
  0.14928330280134472,
  0.01892166851341026,
  0.46070884645826127,
  0.3045205291548069,
  0.3088549594831491,
  0.023774230844750277,
  0.0956786175679702,
  0.4510423180544796,
  0.36831266378265815,
  0.7039638532031971,
  0.2522901868125291,
  0.39550841221268984,
  0.16416781544992987,
  0.06612877303575382,
  0.8750954692178832,
  0.13014575783003968,
  0.003990839806965796,
  0.3787718573868467,
  0.11030324676905545,
  0.0024848567699033985,
  0.09629246709923056,
  0.18685887399468615,
  0.8947584768136979,
  0.691297868360791,
  0.2973735347163676,
  0.6712137966938261,
  0.8039077364748289],
 [0.831639990103628,
  0.06706365429404237,
  0.2781816957080825,
  0.8165524732044934,
  0.04224248218076665,
  0.5763894429654685,
  0.48523260271911095,
  0.1414322431135121,
  0.760996268335222,
  0.7114232994857872,
  0.06188166274905915,
  0.23643409618561884,
  0.23536302560922895,
  0.23151880266252978,
  0.4307800523016173,
  0.58996718826123,
  0.3658251511628502,
  0.250549393126397,
  0.8235484098605914,
  0.6406231633645705,
  0.19611225269822974,
  0.16444398275275418,
  0.280367847698195,
  0.34594762771963516,
  0.015307538895878658,
  0.8399896113159443,
  0.2086044840982636,
  0.40314051461054523,
  0.006024616003876715,
  0.3556095330648904],
 [0.10885736257985934,
  0.05313172413351197,
  0.3968256861461874,
  0.4769501540677896,
  0.02131736345232138,
  0.3790635721099552,
  0.032292960091913736,
  0.3227552688136769,
  0.1914509597379953,
  0.7096420362382898,
  0.5097673593208767,
  0.07157224990580924,
  0.8621130776179804,
  0.6376132279324348,
  0.11474219986094758,
  0.018422558414154855,
  0.7817009501168144,
  0.544213307840334,
  0.2508387105643257,
  0.4485097019222868,
  0.3566496383321468,
  0.5630521939088302,
  0.4205212696249183,
  0.10964702512638225,
  0.2873974948050694,
  0.0657666254468624,
  0.5000239224924629,
  0.4487828600082584,
  0.5019720345860454,
  0.8954387981345988],
 [0.11234224664624537,
  0.38528622775464094,
  0.9003020205137365,
  0.1548403963512751,
  0.20742479979788842,
  0.3148560082450371,
  0.07510838190278712,
  0.36640695182756566,
  0.07766978346354758,
  0.9989311459433597,
  0.13179310229513763,
  0.017204024987458655,
  0.021723227056170358,
  0.0723318836704151,
  0.3918032475554457,
  0.7604600632685691,
  0.07659621307946385,
  0.6546628707502287,
  0.36290277938640536,
  0.3700206676869444,
  0.813644058278554,
  0.23446267024092737,
  0.14308649838271725,
  0.19301452959896392,
  0.8614611170115574,
  0.21300680139500705,
  0.3902700928288624,
  0.8617720316453898,
  0.49496353134195137,
  0.08988839537470139],
 [0.5277561236665556,
  0.1909088700585848,
  0.05494594605641623,
  0.909777452377858,
  0.3845163083806644,
  0.14784907134895042,
  0.07412176976154805,
  0.05527260060659922,
  0.12030771365551984,
  0.780866439288867,
  0.3227369546649297,
  0.26311465697280934,
  0.14277870452168906,
  0.6817852790533395,
  0.34180922983210027,
  0.14210982867089572,
  0.10435399511491161,
  0.9936836841684717,
  0.014767412525204415,
  0.20446014551754732,
  0.9652139584568655,
  0.6979345284266344,
  0.1507697660744184,
  0.3089104642461994,
  0.2525956013402201,
  0.2713438920854101,
  0.17030112187242497,
  0.9708556034357056,
  0.14561276169588433,
  0.20022058166150558],
 [0.7091118165867334,
  0.07841452204447263,
  0.41716174575418097,
  0.16606116564887363,
  0.0262272745448886,
  0.023412485657686832,
  0.4466621896833427,
  0.02969247604594727,
  0.04240789672083356,
  0.7045374447241521,
  0.07081427650586658,
  0.9774189301376537,
  0.4769962670346859,
  0.28570811377641736,
  0.04931802355726789,
  0.31938722832258426,
  0.6709348226784158,
  0.15054050778744377,
  0.35172354929438243,
  0.6238703568122732,
  0.8533559769839717,
  0.24788777271141804,
  0.231280291732606,
  0.06075458780688536,
  0.03902913316017376,
  0.37973546956168325,
  0.6649374320177214,
  0.6448161213704267,
  0.474057220460525,
  0.8499874332397063],
 [0.9051652043525529,
  0.19688878082029737,
  0.40678552885497865,
  0.13343749427518325,
  0.051210815546950506,
  0.07093351399158665,
  0.8136477944075804,
  0.2493952562587476,
  0.1657927313533295,
  0.43221187276319484,
  0.052148879493130185,
  0.4122015517569454,
  0.19029712460749326,
  0.26884617947818534,
  0.4863760356782959,
  0.40806620472779176,
  0.00920888911898421,
  0.6366334934008385,
  0.8722572272360788,
  0.543242835538187,
  0.22201783666534847,
  0.28046672718438714,
  0.9928629271353149,
  0.5918129605122205,
  0.11114198428203137,
  0.11166065246649703,
  0.8956274232529411,
  0.14013794296119608,
  0.5704330442382215,
  0.26038165829494997],
 [0.016140271372693027,
  0.4356863065616663,
  0.27158503312018345,
  0.7787347898130382,
  0.8002296957765616,
  0.5026185510826007,
  0.6245791335446911,
  0.004798936982307711,
  0.163390154644519,
  0.006604508130351738,
  0.9000682602693407,
  0.500680772532488,
  0.6077621021625393,
  0.5694378547238358,
  0.237366063783366,
  0.05244058122806028,
  0.3895029266251683,
  0.608601793177189,
  0.0041963672540458186,
  0.23660384869410556,
  0.34403685137336193,
  0.27335929755932464,
  0.43766793762987477,
  0.5067538399126078,
  0.4112477582169367,
  0.5581238472120774,
  0.4704305456087957,
  0.3843923931235048,
  0.09230192536114024,
  0.2996585128914554],
 [0.18698749964321515,
  0.4598232991178721,
  0.20526249676844244,
  0.2614321416456947,
  0.0008397445239310963,
  0.35794074436821977,
  0.1259459060483431,
  0.7459270931732589,
  0.34293878496090846,
  0.4475472631717585,
  0.5972471096112396,
  0.04072157392163511,
  0.2694945492968306,
  0.5203675248013521,
  0.03641703098606186,
  0.42999168072357363,
  0.09920767196127223,
  0.18871429402712192,
  0.4883299099191384,
  0.45121988084841624,
  0.11358290115579106,
  0.9645251822591664,
  0.7377269113532048,
  0.11435664786827882,
  0.5698136299346147,
  0.40529085464160564,
  0.26755251907792954,
  0.15703956019882304,
  0.02553421462209715,
  0.5842162608412161],
 [0.1797377811203359,
  0.05597623247958983,
  0.49078594769196715,
  0.9303822626082562,
  0.045360810837970944,
  0.11834287009694355,
  0.2425425831027911,
  0.4922835391369019,
  0.26924181055498986,
  0.3024602124519561,
  0.18303657551988373,
  0.7571051792077345,
  0.5590955584920773,
  0.02324053772574808,
  0.7173190382272213,
  0.4718265998617579,
  0.06963005788231624,
  0.03195136211379901,
  0.5126759205596791,
  0.12363719919220229,
  0.19723838588233622,
  0.7110429510291102,
  0.19647963226444198,
  0.2926005403570108,
  0.3827247793010703,
  0.23016606823766905,
  0.9045630449361637,
  0.34534001613876786,
  0.7964983655222303,
  0.16217855235998646],
 [0.24064858325688399,
  0.08639173961675384,
  0.10560821409786414,
  0.040494318771003845,
  0.10630479371718748,
  0.41371278483465335,
  0.14213685896119677,
  0.21524626508084888,
  0.7743575266943573,
  0.18262337959900166,
  0.034289945709560676,
  0.3946991707374279,
  0.15769247432033429,
  0.3472474556870938,
  0.20560980143515173,
  0.7018651678944717,
  0.21143297178576836,
  0.2699876480936022,
  0.752190804050303,
  0.528234619475096,
  0.07614456559415572,
  0.5179295714978497,
  0.747554729116183,
  0.4224024465714118,
  0.5206340872987564,
  0.028243430983315387,
  0.1299802151406031,
  0.6414039376308978,
  0.34598717222772557,
  0.49957467848645876],
 [0.6801297666384484,
  0.5141139340993328,
  0.39354321089726263,
  0.28792965523760405,
  0.5539378911849419,
  0.20383248196586545,
  0.5249900251993088,
  0.18538185295555487,
  0.10590428074343938,
  0.06335144499388645,
  0.7329685924361137,
  0.34370570465924266,
  0.44658304830980766,
  0.19089919654137955,
  0.6058988816957537,
  0.5285846076283026,
  0.25204998071465645,
  0.08804482771083666,
  0.28119944789002604,
  0.40383172736401907,
  0.6898730994038508,
  0.15389653137977033,
  0.7423933742763986,
  0.0956912936493477,
  0.4549610492496987,
  0.05210480381742921,
  0.12252865552904246,
  0.39519659131337415,
  0.2312169351007205,
  0.7750067431662384]]
```

[92]:
```
im_ani = animation.ArtistAnimation(fig, ims, interval=50, repeat_delay=3000,␣
 ↪blit=True)
HTML(im_ani.to_jshtml())
```

[92]:
```
<IPython.core.display.HTML object>
```

[93]:
```
result = nsgaii()
```

[94]:
```
im_ani = animation.ArtistAnimation(fig, ims, interval=50, repeat_delay=3000,␣
 ↪blit=True)
HTML(im_ani.to_jshtml())
```

[94]:
```
<IPython.core.display.HTML object>
```

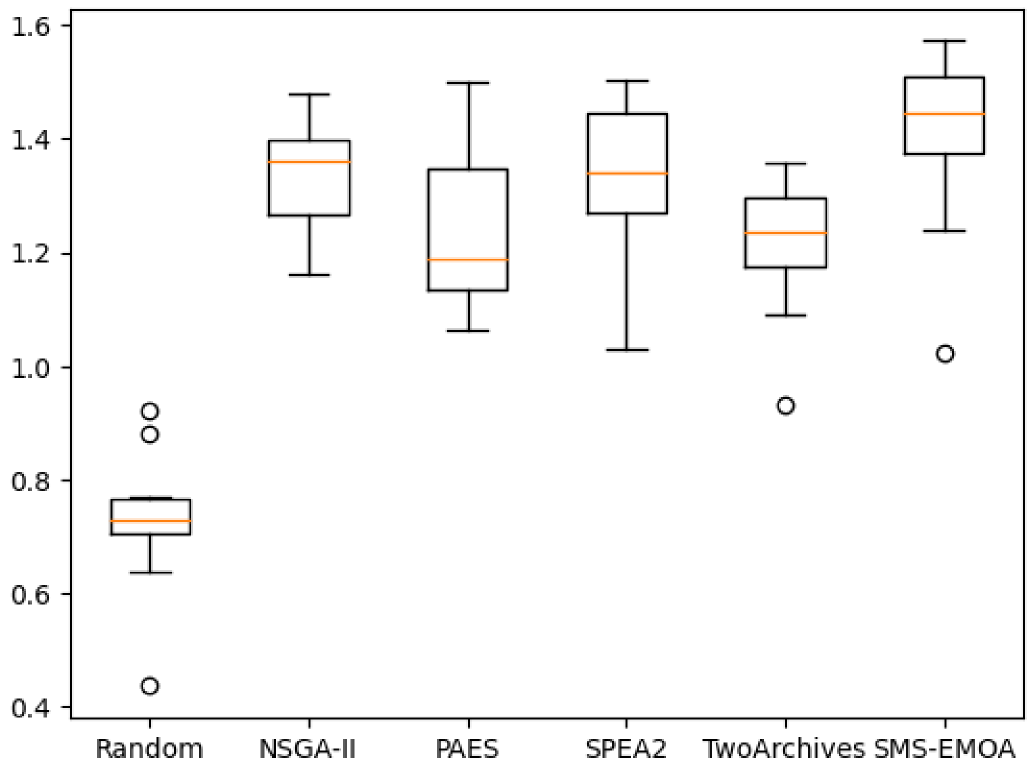
[95]:
```
results = {}

fronts_random = run_times(random_moo, 10)
fronts_nsgaii = run_times(nsgaii, 10)
fronts_paes = run_times(paes, 10)
fronts_spea2 = run_times(spea2, 10)
fronts_ta = run_times(twoarchives, 10)
fronts_sms = run_times(sms, 10)

results["Random"]    = [hypervolume(front, (max_fitness1,max_fitness2)) for␣
 ↪front in fronts_random]
```

```
results["NSGA-II"]  = [hypervolume(front, (max_fitness1,max_fitness2)) for␣
  ↪front in fronts_nsgaii]
results["PAES"] = [hypervolume(front, (max_fitness1,max_fitness2)) for front in␣
  ↪fronts_paes]
results["SPEA2"] = [hypervolume(front, (max_fitness1,max_fitness2)) for front␣
  ↪in fronts_spea2]
results["TwoArchives"] = [hypervolume(front, (max_fitness1,max_fitness2)) for␣
  ↪front in fronts_ta]
results["SMS-EMOA"] = [hypervolume(front, (max_fitness1,max_fitness2)) for␣
  ↪front in fronts_sms]

fig, ax = plt.subplots()
ax.boxplot(results.values())
ax.set_xticklabels(results.keys());
```



For ZDT4, f_1 remains the same, but $f_2 = g(x)[1 - \sqrt{x_1/g(x)}]$, where $g(x) = 1 + 10(n-1) + \Sigma_{i=2}^{n}[x_i^2 - 10cos(4\pi x_i)]$. This results in a highly multi-modal fitness landscape.

[96]:
```
from math import cos, pi

def function2(individual):
    x0 = individual[0]
    g = 0
    for x in individual[1:]:
        g += pow(x, 2) - 10.0 * cos(4.0 * pi * x / 180.0)

    g += 1.0 + 10.0 * (len(individual) - 1)

    return g * (1.0 - sqrt(x0 / g))
```

[97]:
```
max_fitness1 = function1([1 for _ in range(n)])
max_fitness2 = function2([1 for _ in range(n)])
```

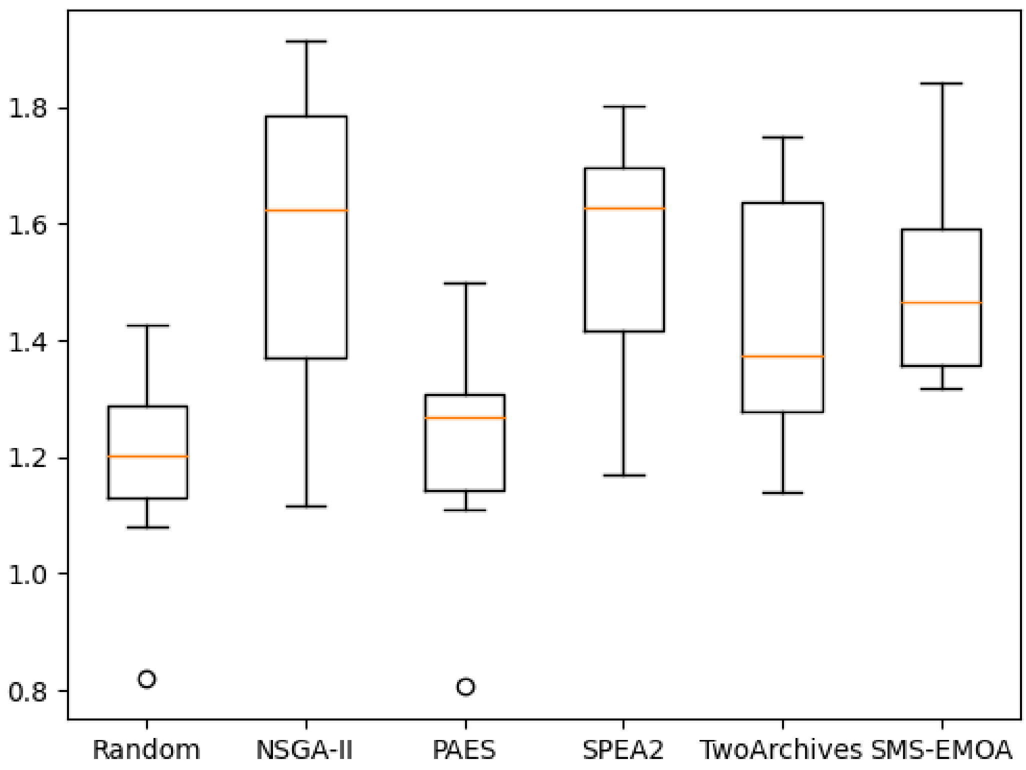
[98]:
```
results = {}

fronts_random = run_times(random_moo, 10)
fronts_nsgaii = run_times(nsgaii, 10)
fronts_paes = run_times(paes, 10)
fronts_spea2 = run_times(spea2, 10)
fronts_ta = run_times(twoarchives, 10)
fronts_sms = run_times(sms, 10)

results["Random"]   = [hypervolume(front, (max_fitness1,max_fitness2)) for␣
 ↪front in fronts_random]
results["NSGA-II"]  = [hypervolume(front, (max_fitness1,max_fitness2)) for␣
 ↪front in fronts_nsgaii]
results["PAES"] = [hypervolume(front, (max_fitness1,max_fitness2)) for front in␣
 ↪fronts_paes]
results["SPEA2"] = [hypervolume(front, (max_fitness1,max_fitness2)) for front␣
 ↪in fronts_spea2]
results["TwoArchives"] = [hypervolume(front, (max_fitness1,max_fitness2)) for␣
 ↪front in fronts_ta]
results["SMS-EMOA"] = [hypervolume(front, (max_fitness1,max_fitness2)) for␣
 ↪front in fronts_sms]

fig, ax = plt.subplots()
ax.boxplot(results.values())
ax.set_xticklabels(results.keys())
```

[98]:
```
[Text(1, 0, 'Random'),
 Text(2, 0, 'NSGA-II'),
 Text(3, 0, 'PAES'),
 Text(4, 0, 'SPEA2'),
 Text(5, 0, 'TwoArchives'),
 Text(6, 0, 'SMS-EMOA')]
```


1.8
1.6
1.4
1.2
1.0
0.8
Random
NSGA-II
PAES
SPEA2
TwoArchives
SMS-EMOA