Multidimensional Knapsack Problem

Using a Genetic Algorithm

Nicolò Vescera

Introduzione

Problema

$$max \sum_{j=1}^{n} p_j x_j$$

$$\text{subject to:} \sum_{j=1}^{n} r_{i,j} x_j \leq b_i, \quad i=1,2,\ldots,m$$

Soluzione

$$items = [0, 0, 1, 1, 1, 0, 1, 0]$$

$$fitness = 3980.21$$

Istanze del Problema

```
// MK01.TXT
6
100,600,1200,2400,500,2000
10
8,12,13,64,22,41
8,12,13,75,22,41
3,6,4,18,6,4
5,10,8,32,6,12
5,13,8,42,6,20
5,13,8,48,6,20
0,0,0,0,8,0
3,0,4,0,8,0
3,2,4,0,8,4
3,2,4,8,8,4
80,96,20,36,44,48,10,18,22,24
3800
```

Oggetti

Valori

Numero Vincoli

Parametri

Vincoli

Soluzione

Istanze del Problema

ID	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	Value
0	8.0	8.0	3.0	5.0	5.0	5.0	0.0	3.0	3.0	3.0	100.0
1	12.0	12.0	6.0	10.0	13.0	13.0	0.0	0.0	2.0	2.0	600.0
2	12.0	13.0	4.0	8.0	8.0	8.0	0.0	4.0	4.0	4.0	1200.0
3	64.0	75.0	18.0	32.0	42.0	48.0	0.0	0.0	0.0	8.0	2400.0
4	22.0	22.0	6.0	6.0	6.0	6.0	8.0	8.0	8.0	8.0	500.0
5	41	41.0	4.0	12.0	20.0	20.0	0.0	0.0	4.0	4.0	2000.0

Implementazione

Initial Population

```python
def init_population(self):
    self.population: List[Solution] = []
    self.f_obj: List[float] = list(np.zeros(self.num_elem))
    self.best: Solution = None
    self.best_f: float = float('-inf')  # tiny number

    for i in range(self.num_elem):
        tmp_sol: Solution = np.zeros(self.num_items)

        # list of indexes (e.g. 1 means df[1])
        T: List[int] = list(range(self.num_items))

        # temporary actual constrints sum
        R = np.zeros(len(self.problem.W))

        # randomly extract an item
        j = T.pop(random.randint(0, len(T) - 1))
        item = self.problem.df.loc[:, self.problem.df.columns !=
                                   'Value'].loc[j].to_numpy()

        # try to add extracted item, then extract a new one and so on
        while all(R + item <= self.problem.W):
            tmp_sol[j] = 1

            R = R + item

            # no more items left, continue to new solution
            if len(T) <= 0:
                break

            j = T.pop(random.randrange(len(T)))
            item = self.problem.df.loc[:, self.problem.df.columns !=
                                       'Value'].loc[j].to_numpy()

    self.population.append(tmp_sol)
    self.f_obj[i] = self.problem.objective_function(tmp_sol)
    self.update_best(tmp_sol, self.f_obj[i], 0)
```

Mating Pool Selection

```python
def select_mating_pool(self) -> List[Tuple[Solution, Solution]]:

    def tournament(k: int = 5) -> Solution:
        random_select_solutions = [
                random.randint(0,
                               len(self.population) - 1) for _ in range(k)
              ]

        # generate a dictioray {solution index : solution fitness}
        # e.g. solution 1/16 has fitness vale of 287
        selected_objectivefunctions = {
            i: self.f_obj[i]
                for i in random_select_solutions
        }

        # find solution index with max fitless value
        # e.g. solution 3/16 has the max fintess value
        max_index = max(selected_objectivefunctions,
                            key=selected_objectivefunctions.get)

        return self.population[max_index]

    mating_pool = []

    for i in range(len(self.population) // 2):
        c1 = tournament()
        c2 = tournament()
        mating_pool.append((
                c1,
                c2,
        ))

    return mating_pool
```

Crossover Operator

```python
def do_crossover(self, mating_pool: List[Tuple[Solution,Solution]])
                                                    -> List[Solution]:

    def uniform_crossover_operator(s1: Solution, s2: Solution)
                                                    -> Solution:
        '''
        From parents (s1 and s2) generate only 1 child (c) using
        a random probability to choose a chromosome from s1 or s2
        '''
        c = np.zeros(len(s1))

        for i in range(len(s1)):
            # random True or False.
            # faster than `random.choice([True, False])`
            c[i] = s1[i] if bool(random.getrandbits(1)) else s2[i]

        return c

    children = []

    for s1, s2 in mating_pool:
        if random.random() < self.pcross:
            c = uniform_crossover_operator(s1, s2)
            children.append(c)

            continue

        children.append(s1)
        children.append(s2)

    return children
```

Mutation Operator

```python
def do_mutation(self, children: List[Solution]):
    '''
    Randomly flip bits according to pmut probability
    '''
    for child in children:
        for i in range(len(child)):
            if random.random() < self.pmut:
                child[i] = int(not child[i])
```

Repair Operator

```
def repair_operator(self, children: List[Solution]) -> Solution:
    for child in children:
        child_parameters = self.problem.df.loc[:,
                               self.problem.df.columns != 'Value'].
                                           loc[child ==1].sum().to_numpy()
        # good child, check next child
        if all(child_parameters <= self.problem.W):
            continue

        old_child = child.copy()

        # DROP PHASE
        i = 0
        while any(child_parameters > self.problem.W):
            # delete item from solution
            child[self.problem.sorted_value_objects_indexes[i]] = 0

            # update parameters
            child_parameters = self.problem.df.loc[:,
                                   self.problem.df.columns != 'Value'].loc[
                                               child == 1].sum().to_numpy()
            i = i + 1

        # ADD PAHSE
        for i in reversed(range(len(child))):
            # temporary edited child
            tmp_child = child.copy()

            # add item to solution
            tmp_child[self.problem.sorted_value_objects_indexes[i]] = 1

            # update parameters
            tmp_child_parameters = self.problem.df.loc[:,
                                       self.problem.df.columns !='Value'].loc[
                                                tmp_child == 1].sum().to_numpy()
            # update child with tmp mods
            # if is feasible solution
            if all(tmp_child_parameters <= self.problem.W):
                child = tmp_child.copy()
                child_parameters = tmp_child_parameters

```

Select New Population

```python
def select_new_population(self, children: List[Solution], gen: int):

    def select_best():
        total_solutions: List[Solution] = self.population + children
        total_fintesses: List[float] = self.f_obj + [
            self.problem.objective_function(c) for c in children
        ]

        assert len(total_solutions) == len(total_fintesses)

        total_indexes: List[int] = list(range(len(total_solutions)))
        total_indexes.sort(key=lambda i: total_fintesses[i],
                                                   reverse=True)
        best_indexes: List[int] = total_indexes[:self.num_elem]

        self.population = [total_solutions[i] for i in best_indexes]
        self.f_obj = [total_fintesses[i] for i in best_indexes]

        self.update_best(self.population[0], self.f_obj[0], gen)

    select_best()
```

Tuning Phase

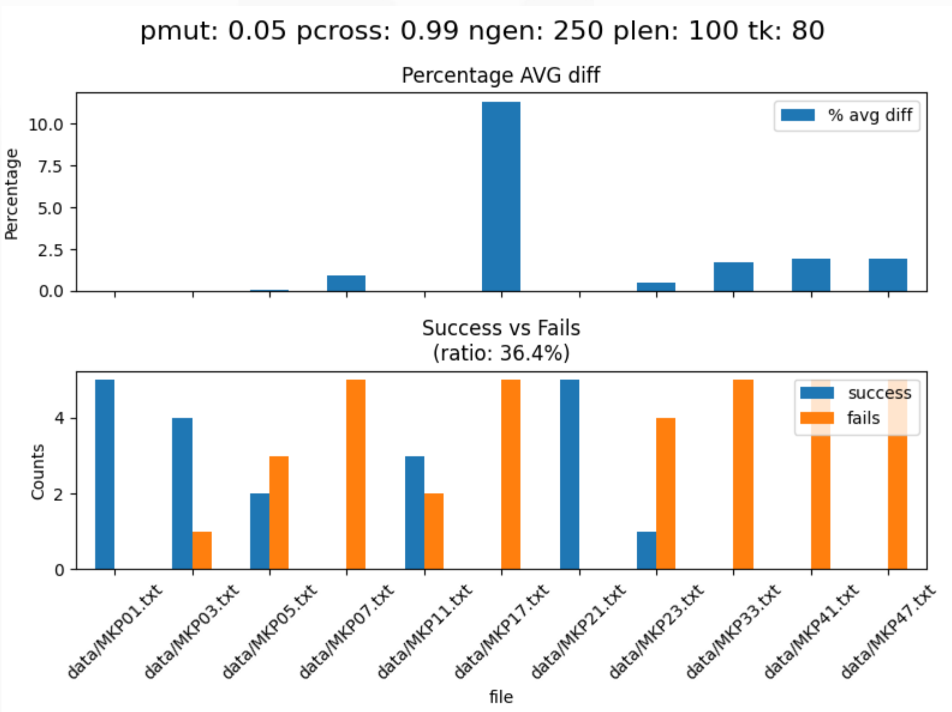
Tuning Dataset

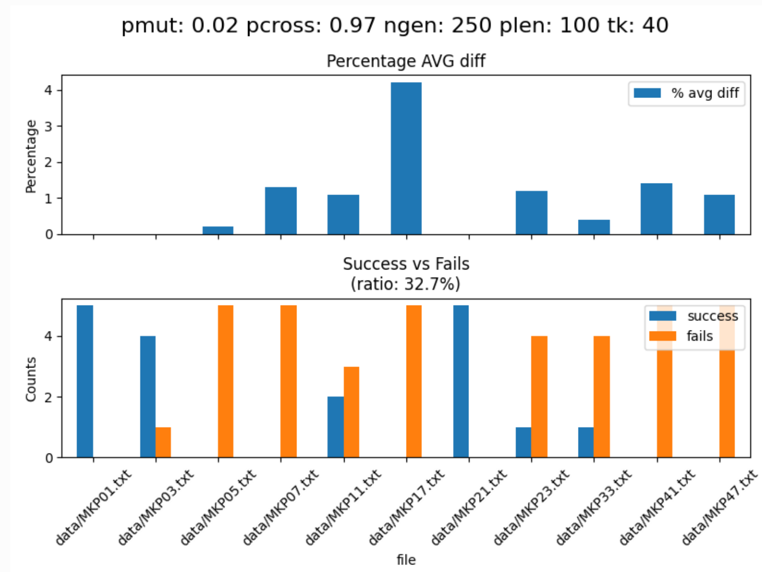
File	Oggetti	Vincoli
MKP01.txt	6	10
MKP03.txt	15	10
MKP05.txt	28	10
MKP07.txt	50	5
MKP11.txt	28	2
MKP17.txt	105	2
MKP21.txt	30	5
MKP23.txt	40	5
MKP33.txt	60	5
MKP41.txt	80	5
MKP47.txt	90	5

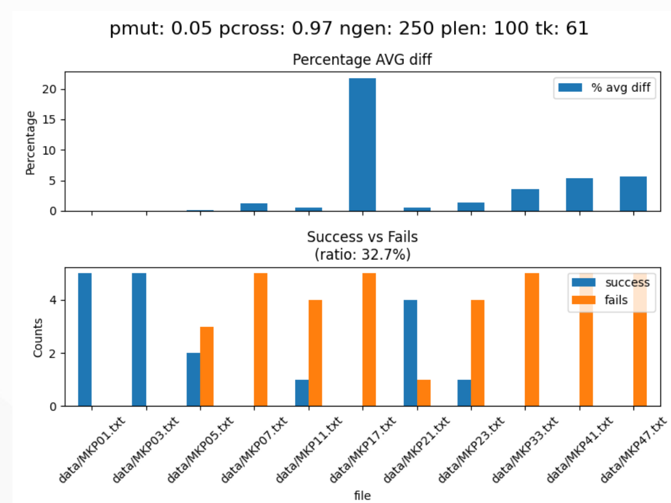
Output

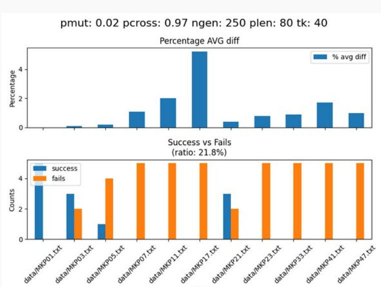
```
,file,gen,found,target,success,diff
0,data/MKP01.txt,52,3800.0,3800.0,True,0.0
1,data/MKP01.txt,1,3800.0,3800.0,True,0.0
2,data/MKP01.txt,0,3800.0,3800.0,True,0.0
3,data/MKP01.txt,48,3800.0,3800.0,True,0.0
4,data/MKP01.txt,87,3800.0,3800.0,True,0.0
5,data/MKP03.txt,20,3915.0,4015.0,False,100.0
6,data/MKP03.txt,76,3525.0,4015.0,False,490.0
7,data/MKP03.txt,81,3850.0,4015.0,False,165.0
8,data/MKP03.txt,44,3965.0,4015.0,False,50.0
9,data/MKP03.txt,0,3820.0,4015.0,False,195.0
10,data/MKP05.txt,16,11880.0,12400.0,False,520.0
11,data/MKP05.txt,90,11940.0,12400.0,False,460.0
```

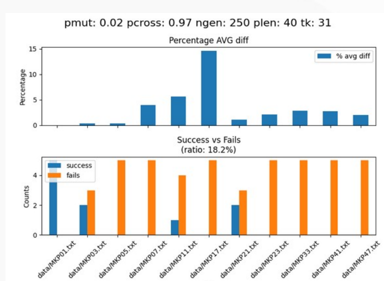
Risultati

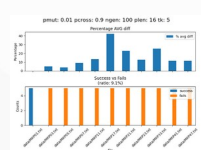

pmut: 0.05 pcross: 0.99 ngen: 250 plen: 100 tk: 80
Percentage AVG diff
% avg diff
Percentage
Success vs Fails
(ratio: 36.4%)
success
fails
Counts
data/MKP01.txt
data/MKP03.txt
data/MKP05.txt
data/MKP07.txt
data/MKP11.txt
data/MKP17.txt
data/MKP21.txt
data/MKP23.txt
data/MKP33.txt
data/MKP41.txt
data/MKP47.txt
file

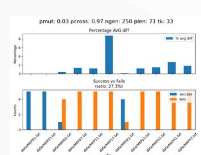

pmut: 0.02 pcross: 0.97 ngen: 250 plen: 100 tk: 40
Percentage AVG diff
% avg diff
Percentage
Success vs Fails
(ratio: 32.7%)
success
fails
Counts
file


pmut: 0.05 pcross: 0.97 ngen: 250 plen: 100 tk: 61
Percentage AVG diff
% avg diff
Percentage
Success vs Fails
(ratio: 32.7%)
success
fails
Counts
file


pmut: 0.02 pcross: 0.97 ngen: 250 plen: 80 tk: 40
Percentage AVG diff
Success vs Fails
(ratio: 21.8%)


pmut: 0.02 pcross: 0.97 ngen: 250 plen: 40 tk: 31
Percentage AVG diff
Success vs Fails
(ratio: 18.2%)





Top 3

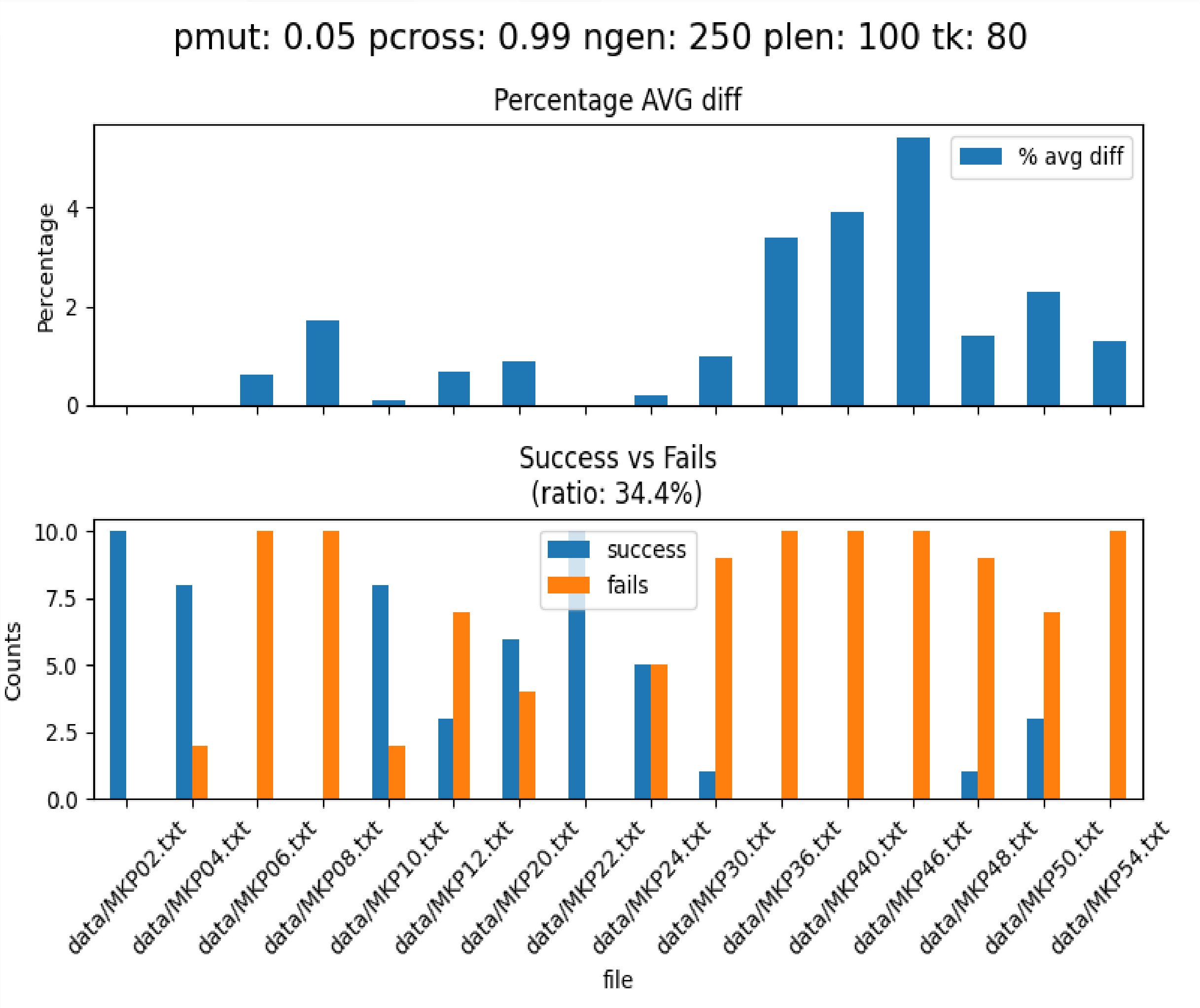
pmut	pcross	ngen	plen	tk	Success Ratio (%)	Max AVG Diff (%)
0.05	0.99	250	100	80	36.4	12.0
0.02	0.97	250	100	41	32.7	5.0
0.05	0.97	250	100	61	32.7	21.0

Testing Phase

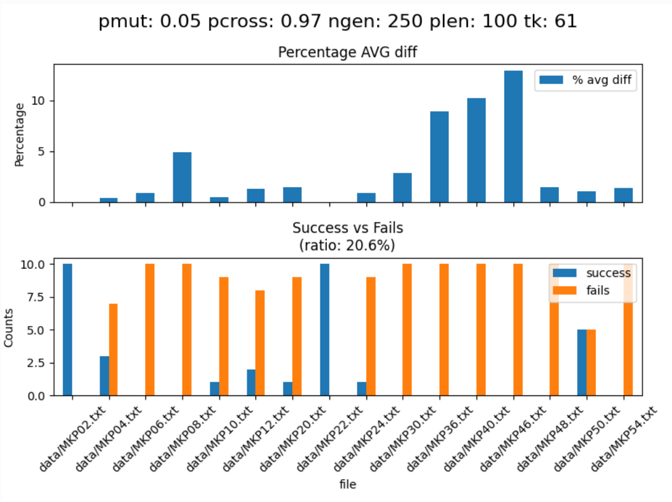
Testing Dataset

File	Oggetti	Vincoli
MKP02.txt	10	10
MKP04.txt	20	10
MKP06.txt	39	5
MKP08.txt	60	30
MKP10.txt	28	2
MKP12.txt	28	2
MKP20.txt	30	5
MKP22.txt	30	5
MKP24.txt	40	5
MKP30.txt	50	5
MKP36.txt	70	5
MKP40.txt	80	5
MKP46.txt	90	5
MKP48.txt	27	4
MKP50.txt	29	2
MKP54.txt	28	4

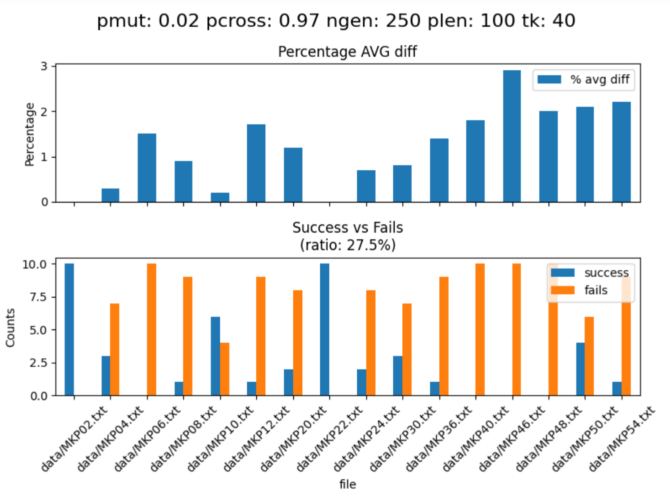
Risultati



pmut: 0.05 pcross: 0.99 ngen: 250 plen: 100 tk: 80
Percentage AVG diff
% avg diff
Percentage
Success vs Fails
(ratio: 34.4%)
success
fails
Counts
data/MKP02.txt
data/MKP04.txt
data/MKP06.txt
data/MKP08.txt
data/MKP10.txt
data/MKP12.txt
data/MKP20.txt
data/MKP22.txt
data/MKP24.txt
data/MKP30.txt
data/MKP36.txt
data/MKP40.txt
data/MKP46.txt
data/MKP48.txt
data/MKP50.txt
data/MKP54.txt
file



pmut: 0.05 pcross: 0.97 ngen: 250 plen: 100 tk: 61
Percentage AVG diff
% avg diff
Percentage
Success vs Fails
(ratio: 20.6%)
success
fails
Counts
file



pmut: 0.02 pcross: 0.97 ngen: 250 plen: 100 tk: 40
Percentage AVG diff
% avg diff
Percentage
Success vs Fails
(ratio: 27.5%)
success
fails
Counts
file