

Multidimensional Knapsack Problem

Using a Genetic Algorithm

Introduzione

The background features a dark grey, angular shape that rises from the bottom left towards the top right. To the left of this dark shape, there are several overlapping, semi-transparent light grey triangles of various sizes, creating a layered, abstract effect.

Problema

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

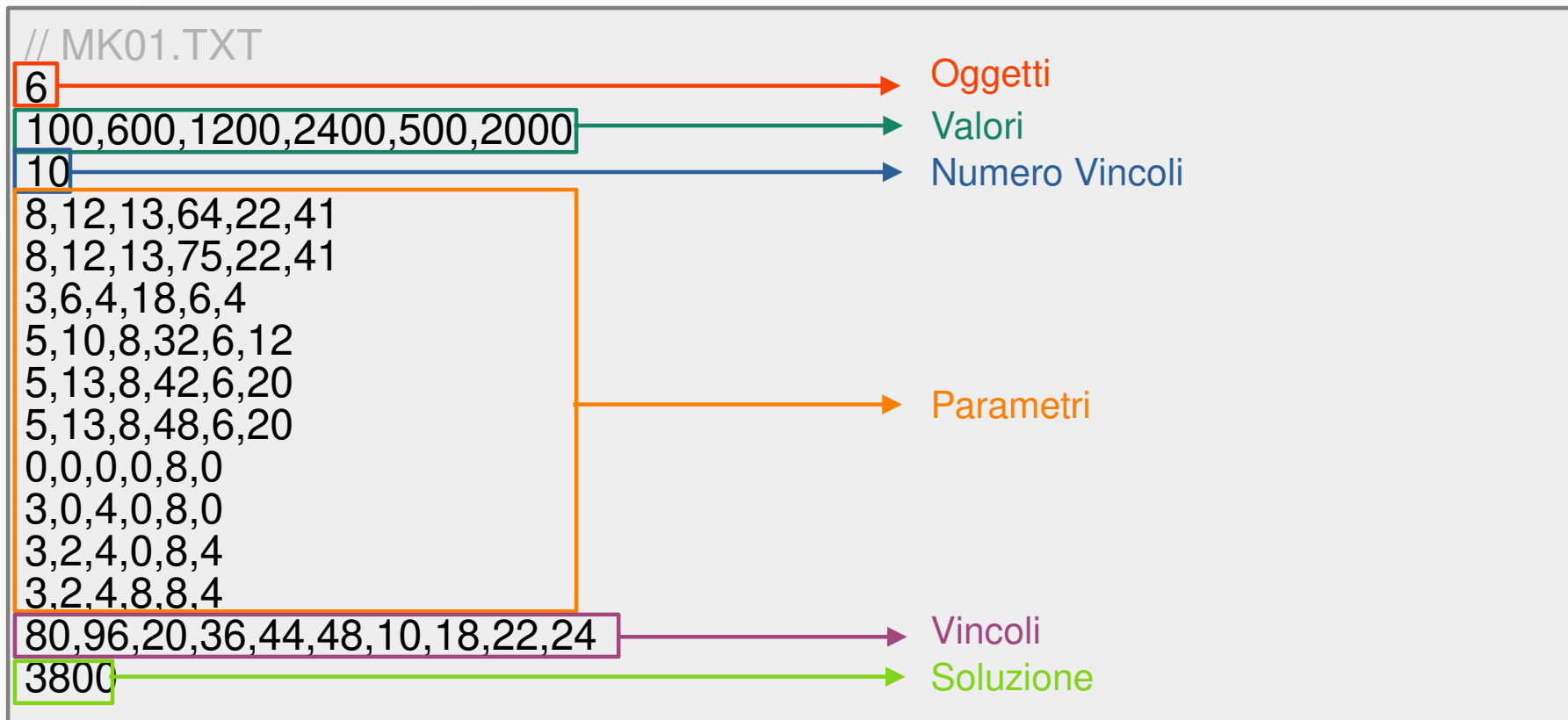
$$\text{subject to: } \sum_{j=1}^n r_{i,j} x_j \leq b_i, \quad i=1, 2, \dots, m$$

Soluzione

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

$fitness = 3980.21$

Istanze del Problema



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

```
1 def init_population(self):
2     self.population: List[Solution] = []
3     self.f_obj: List[float] = list(np.zeros(self.num_elem))
4     self.best: Solution = None
5     self.best_f: float = float('-inf') # tiny number
6
7     for i in range(self.num_elem):
8         tmp_sol: Solution = np.zeros(self.num_items)
9
10        # list of indexes (e.g. 1 means df[1])
11        T: List[int] = list(range(self.num_items))
12
13        # temporary actual constraints sum
14        R = np.zeros(len(self.problem.W))
15
16        # randomly extract an item
17        j = T.pop(random.randint(0, len(T) - 1))
18        item = self.problem.df.loc[:, self.problem.df.columns !=
19                                   'Value'].loc[j].to_numpy()
20
21        # try to add extracted item, then extract a new one and so on
22        while all(R + item <= self.problem.W):
23            tmp_sol[j] = 1
24
25            R = R + item
26
27            # no more items left, continue to new solution
28            if len(T) <= 0:
29                break
30
31            j = T.pop(random.randrange(len(T)))
32            item = self.problem.df.loc[:, self.problem.df.columns !=
33                                   'Value'].loc[j].to_numpy()
34
35        self.population.append(tmp_sol)
36        self.f_obj[i] = self.problem.objective_function(tmp_sol)
37        self.update_best(tmp_sol, self.f_obj[i], 0)
38
39
40
41
```


Mating Pool Selection

```
1 def select_mating_pool(self) → List[Tuple[Solution, Solution]]:
2
3     def tournament(k: int = 5) → Solution:
4         random_select_solutions = [
5             random.randint(0,
6                 len(self.population) - 1) for _ in range(k)
7         ]
8
9         # generate a dictionary {solution index : solution fitness}
10        # e.g. solution 1/16 has fitness value of 287
11        selected_objectivefunctions = {
12            i: self.f_obj[i]
13            for i in random_select_solutions
14        }
15
16        # find solution index with max fitness value
17        # e.g. solution 3/16 has the max fitness value
18        max_index = max(selected_objectivefunctions,
19                        key=selected_objectivefunctions.get)
20
21        return self.population[max_index]
22
23    mating_pool = []
24
25    for i in range(len(self.population) // 2):
26        c1 = tournament()
27        c2 = tournament()
28        mating_pool.append((
29            c1,
30            c2,
31        ))
32
33    return mating_pool
34
35
36
37
38
39
40
41
```

Crossover Operator

```
1 def do_crossover(self, mating_pool: List[Tuple[Solution, Solution]])  
2                                     → List[Solution]:  
3  
4     def uniform_crossover_operator(s1: Solution, s2: Solution)  
5                                     → Solution:  
6         '''  
7         From parents (s1 and s2) generate only 1 child (c) using  
8         a random probability to choose a chromosome from s1 or s2  
9         '''  
10        c = np.zeros(len(s1))  
11  
12        for i in range(len(s1)):  
13            # random True or False.  
14            # faster than `random.choice([True, False])`  
15            c[i] = s1[i] if bool(random.getrandbits(1)) else s2[i]  
16  
17        return c  
18  
19    children = []  
20  
21    for s1, s2 in mating_pool:  
22        if random.random() < self.pcross:  
23            c = uniform_crossover_operator(s1, s2)  
24            children.append(c)  
25  
26            continue  
27  
28        children.append(s1)  
29        children.append(s2)  
30  
31    return children  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41
```

Mutation Operator

```
1  def do_mutation(self, children: List[Solution]):
2      '''
3      Randomly flip bits according to pmut probability
4      '''
5      for child in children:
6          for i in range(len(child)):
7              if random.random() < self.pmut:
8                  child[i] = int(not child[i])
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
```

Repair Operator

```
1 def repair_operator(self, children: List[Solution]) -> Solution:
2     for child in children:
3         child_parameters = self.problem.df.loc[:,
4                               self.problem.df.columns != 'Value'].
5                               loc[child == 1].sum().to_numpy()
6         # good child, check next child
7         if all(child_parameters <= self.problem.W):
8             continue
9
10        old_child = child.copy()
11
12        # DROP PHASE
13        i = 0
14        while any(child_parameters > self.problem.W):
15            # delete item from solution
16            child[self.problem.sorted_value_objects_indexes[i]] = 0
17
18            # update parameters
19            child_parameters = self.problem.df.loc[:,
20                              self.problem.df.columns != 'Value'].loc[
21                                  child == 1].sum().to_numpy()
22
23            i = i + 1
24
25        # ADD PHASE
26        for i in reversed(range(len(child))):
27            # temporary edited child
28            tmp_child = child.copy()
29
30            # add item to solution
31            tmp_child[self.problem.sorted_value_objects_indexes[i]] = 1
32
33            # update parameters
34            tmp_child_parameters = self.problem.df.loc[:,
35                                  self.problem.df.columns != 'Value'].loc[
36                                      tmp_child == 1].sum().to_numpy()
37
38            # update child with tmp mods
39            # if is feasible solution
40            if all(tmp_child_parameters <= self.problem.W):
41                child = tmp_child.copy()
42                child_parameters = tmp_child_parameters
```

Select New Population

```
1 def select_new_population(self, children: List[Solution], gen: int):
2
3     def select_best():
4         total_solutions: List[Solution] = self.population + children
5         total_fintesses: List[float] = self.f_obj + [
6             self.problem.objective_function(c) for c in children
7         ]
8
9         assert len(total_solutions) == len(total_fintesses)
10
11         total_indexes: List[int] = list(range(len(total_solutions)))
12         total_indexes.sort(key=lambda i: total_fintesses[i],
13                             reverse=True)
14         best_indexes: List[int] = total_indexes[:self.num_elem]
15
16         self.population = [total_solutions[i] for i in best_indexes]
17         self.f_obj = [total_fintesses[i] for i in best_indexes]
18
19         self.update_best(self.population[0], self.f_obj[0], gen)
20
21     select_best()
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
```



Tuning Phase

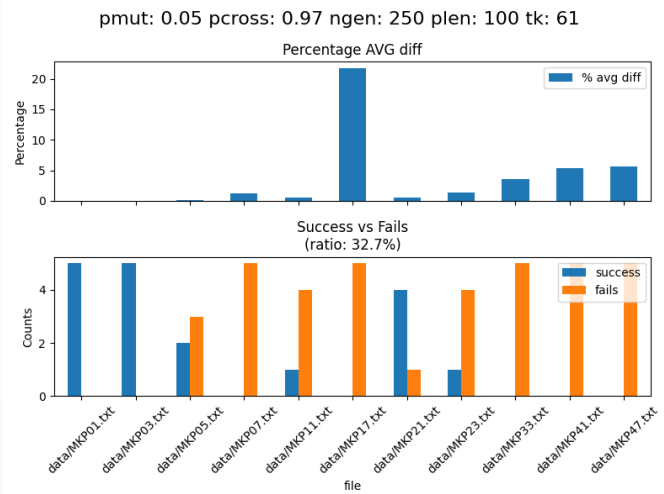
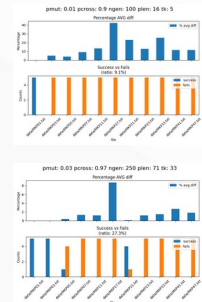
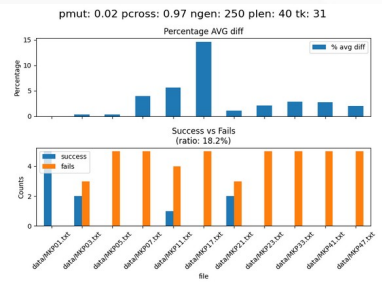
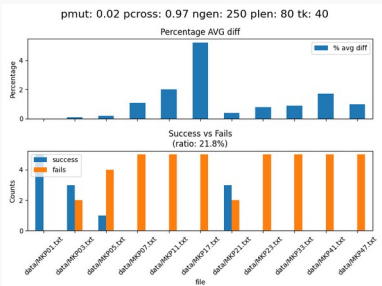
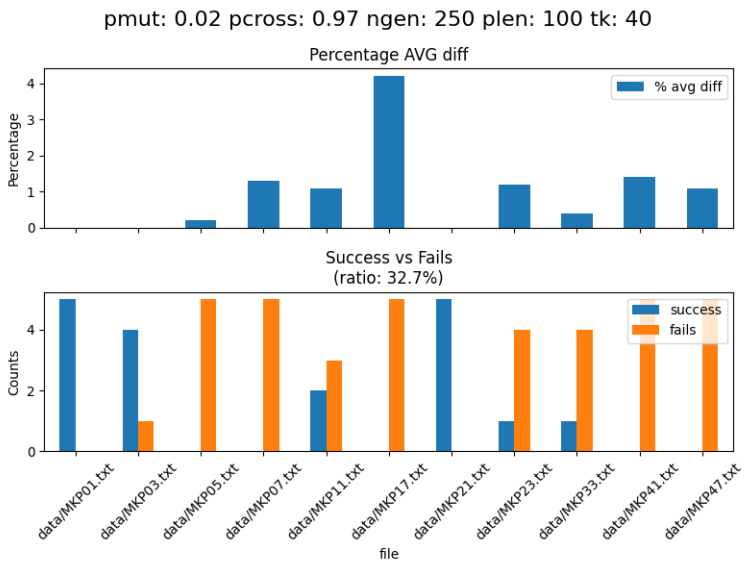
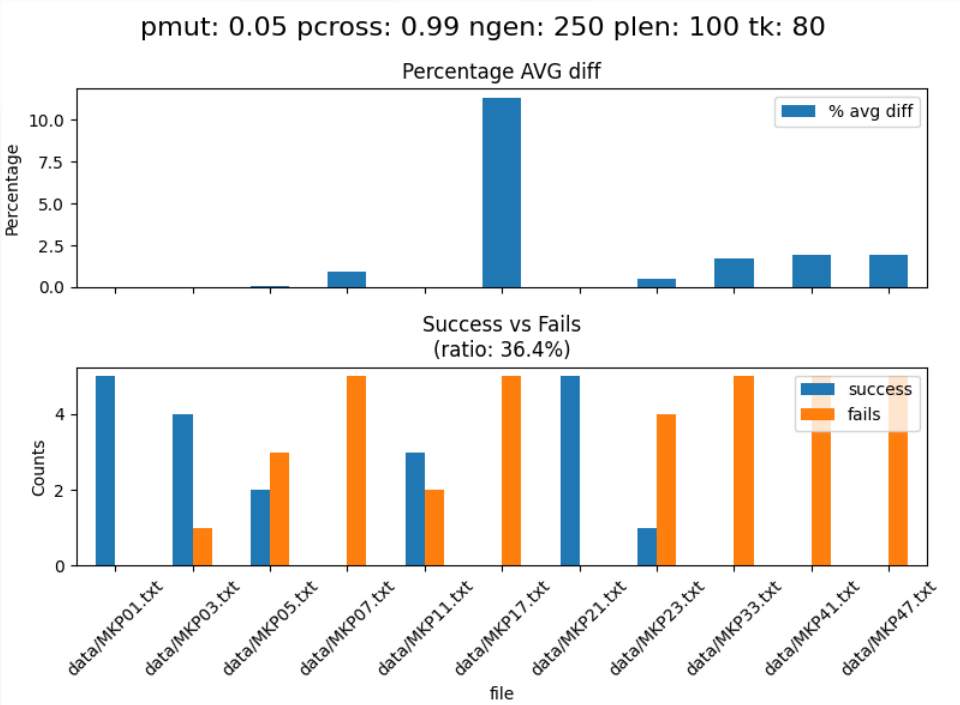
Tuning Dataset

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



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

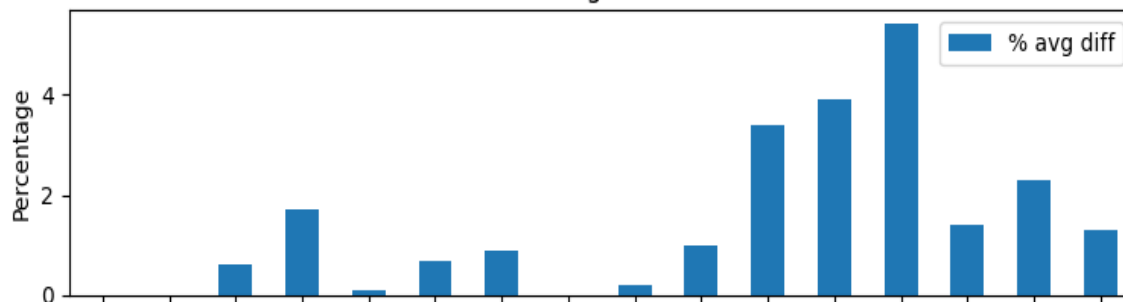
Testing Dataset

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

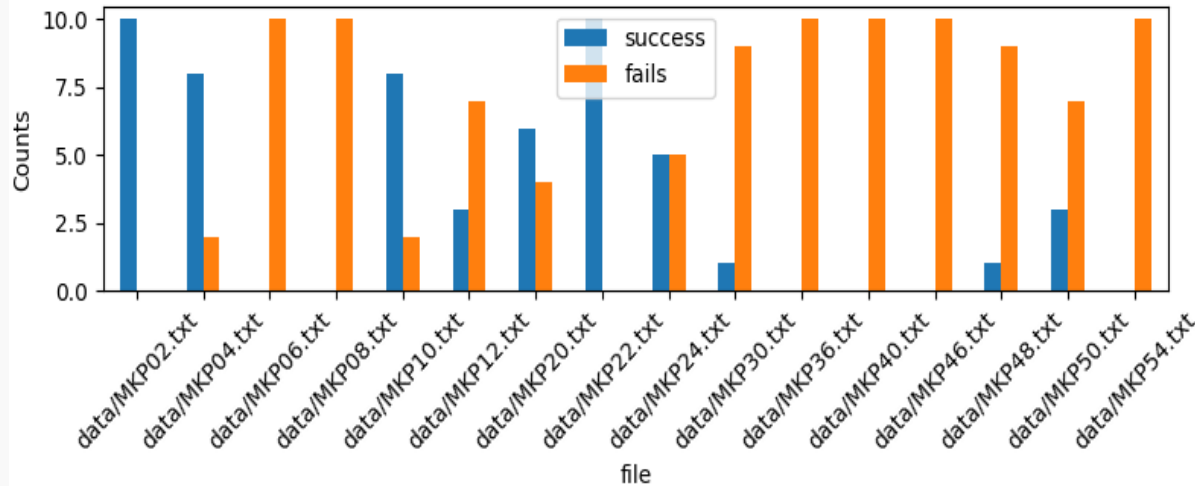
Risultati

pmut: 0.05 pcross: 0.99 ngen: 250 plen: 100 tk: 80

Percentage AVG diff

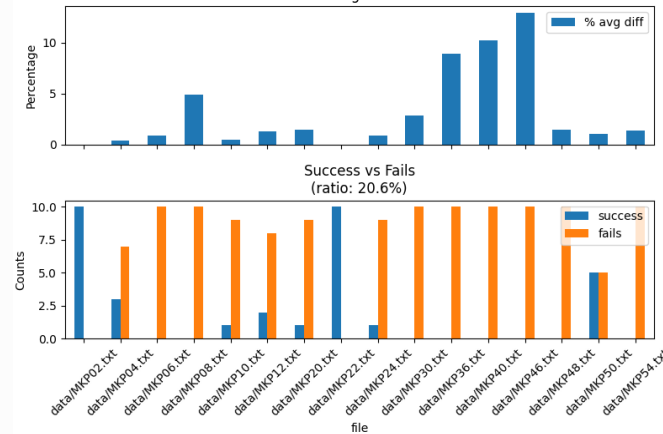


Success vs Fails
(ratio: 34.4%)

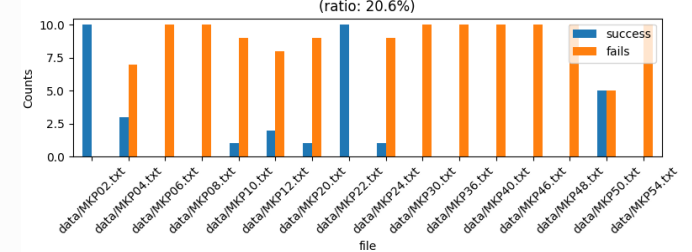


pmut: 0.05 pcross: 0.97 ngen: 250 plen: 100 tk: 61

Percentage AVG diff

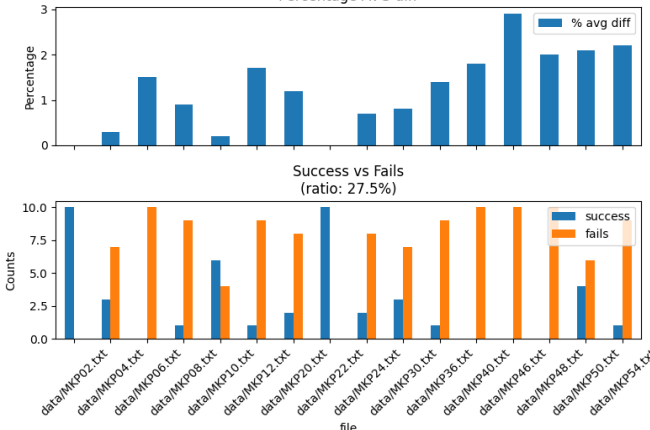


Success vs Fails
(ratio: 20.6%)



pmut: 0.02 pcross: 0.97 ngen: 250 plen: 100 tk: 40

Percentage AVG diff



Success vs Fails
(ratio: 27.5%)

