Python Code

```python
from random import random
from functools import reduce
from collections import namedtuple
from queue import PriorityQueue, SimpleQueue, LifoQueue
import matplotlib.pyplot as plt
import numpy as np

def goal_check(state):
    return np.all(reduce(np.logical_or, [SETS[i] for i in state.
    taken], np.array([False for _ in range(PROBLEM_SIZE)])))

def frequencies(A):
    return np.sum(A, axis=0)

def rank_0(A):
    return np.sum(A, axis=1)



PROBLEM_SIZE = 60
NUM_SETS = 40
SETS = tuple(np.array([random() < .1 for _ in range(PROBLEM_SIZE)])
     for _ in range(NUM_SETS))
State = namedtuple('State', ['taken', 'not_taken'])


numeric_sets = np.array(SETS, dtype=int)



tiles = 0
cs = State(set(), set(range(NUM_SETS)))

while not goal_check(cs):
    tiles += 1
    freq = frequencies(numeric_sets)
    card = rank_0(numeric_sets)
    weights = (100 - freq) * 0.01
    rank = card + np.dot(numeric_sets, weights)
    action = np.argmax(rank, axis=0)

    remove = np.argwhere(numeric_sets[action] == 1)
    numeric_sets = np.delete(numeric_sets, remove, axis=1)
    cs = State(cs.taken ^ {action}, cs.not_taken ^ {action})

print(tiles, cs.taken)
```

The provided Python code is a deterministic heuristic-based algorithm for solving the set covering problem.

ISTANCE GENERATION

```
1 PROBLEM_SIZE = 60
2 NUM_SETS = 40
3 SETS = tuple(np.array([random() < .1 for _ in range(PROBLEM_SIZE)])
      for _ in range(NUM_SETS))
4 State = namedtuple('State', ['taken', 'not_taken'])
5
6
7 numeric_sets = np.array(SETS, dtype=int)
```

At line 7 we create a np.array duplicate of the istance to faster our following operations.

Keep in mind that we have NUM SETS tiles with length PROBLEM SIZE

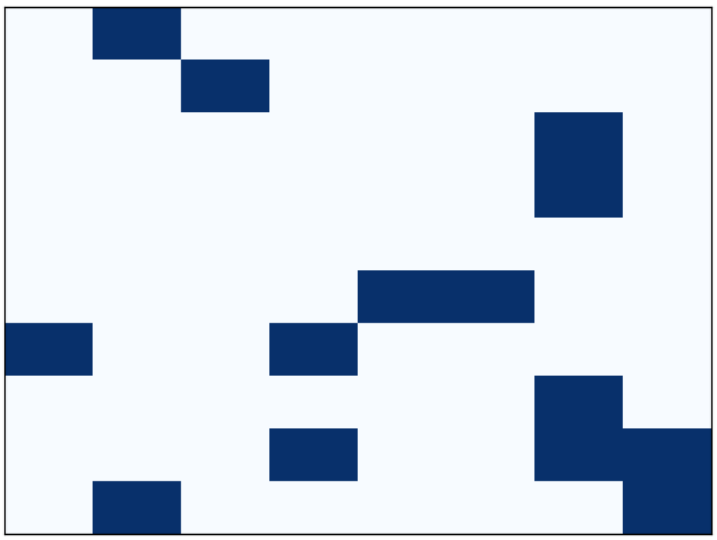

Figure 1: 10 tiles with length 8

RANKING CANDIDATES

```
1 def frequencies(A):
2     return np.sum(A, axis=0)
3
4 def rank_0(A):
5     return np.sum(A, axis=1)
```

To rank different candidates we use

- FREQUENCIES

  summing booleans vertically we get,for every position (horizontally) how many tiles contain it.

- RANK0

  we find the cardinality of the tiles (how many elements they contain)

```
1     freq = frequencies(numeric_sets)
2     card = rank_0(numeric_sets)
3     weights = (100 - freq) * 0.01
4     rank = card + np.dot(numeric_sets, weights)
```

After obtaining the first two indicators from the previously explained functions, at line 3 we get weights inversely proportional to the frequencies (if an element is present in many tiles it is not so important). with weights array we get a sense of the importance of every tile (line 4 np.dot).

We then sum in line 4 this rank to the cardinalities.

```
[1 2 1 2 1 1 4 2]
[1 1 1 1 0 2 2 1 3 2]
[0.99 0.98 0.99 0.98 0.99 0.99 0.96 0.98]
```

Figure 2: frequencies,cardinality and weights for Figure 1 instance

Now we have a sense of the real importance of each tile.

```
[ 4.92  4.96  4.84  4.84  0.    9.92  9.88  4.84 14.68  9.84]
```

Figure 3: final ranks

TAKE ACTION AND GO ON

```
action = np.argmax(rank, axis=0)
```

We choose the highest ranked tile and update both the matrix and the current state (cs):

```
remove = np.argwhere(numeric_sets[action] == 1)
numeric_sets = np.delete(numeric_sets, remove, axis=1)
cs = State(cs.taken ^ {action}, cs.not_taken ^ {action})
```

The remove function is to delete already taken elements from the instance matrix so that the algo does not take them in account
vspace2

CONSIDERATIONS

This heuristic is deterministic, works pretty well also on sparse and large instances. It is guaranteed to converge to a solution

It is also CONSISTENT,meaning that for each step we make the cost we pay (+1 tile) is at least compensated by the fact that we reduce the distance to the objective by at least 1.

Stochastic variant

To try and explore better the states around us we take advantage of a little variation:

```
p = softmax(rank)
action=np.random.choice(len(p), p=p)
```

After ranking tiles,we use the ranks to compute probabilities of choosing them with the softmax operator.

As a matter of fact,if the ranks already show some particularly favorite candidates,the probabilities will lean heavily towards them, because of the softmax mathematical formulation.

```
DET=np.zeros(100)
STOCH=np.zeros(100)
PROBLEM_SIZE =70
NUM_SETS =50
for i in range(100):
  SETS = tuple(np.array([random() < .3 for _ in range(PROBLEM_SIZE)
    ]) for _ in range(NUM_SETS))
  if not goal_check(State(set(range(NUM_SETS)), set())):
    STOCH[i]=0
    DET[i]=0
    print(f"nope at {i}")
  else:
    numeric_sets = np.array(SETS, dtype=int)
    a=numeric_sets #DUMMY VARIABLE TO STORE INFO
    tiles=0
    cs=State(set(), set(range(NUM_SETS)))
    while not goal_check(cs):
      tiles+=1
      freq=frequencies(numeric_sets)
      card=rank_0(numeric_sets)
      weights=(100-freq)*0.01
      rank=card+np.dot(numeric_sets,weights)
      p = softmax(rank)
      action=np.random.choice(len(p), p=p)
      remove=np.argwhere(numeric_sets[action] == 1)
      numeric_sets=np.delete(numeric_sets,remove,axis=1)
      cs=State(cs.taken ^ {action},cs.not_taken ^ {action})
    STOCH[i]=tiles
    numeric_sets=a  #DUMMY VARIABLE IN ACTION
    tiles=0
    cs=State(set(), set(range(NUM_SETS)))
    while not goal_check(cs):
      tiles+=1
      freq=frequencies(numeric_sets)
      card=rank_0(numeric_sets)
      weights=(100-freq)*0.01
      rank=card+np.dot(numeric_sets,weights)
      p = softmax(rank)
      action=np.random.choice(len(p), p=p)
      remove=np.argwhere(numeric_sets[action] == 1)
      numeric_sets=np.delete(numeric_sets,remove,axis=1)
      cs=State(cs.taken ^ {action},cs.not_taken ^ {action})
    DET[i]=tiles
print(STOCH,DET)
print(DET)
print(np.sum(np.array(DET>STOCH),axis=0),np.sum(np.array(STOCH>0),
    axis=0))
```

RESULTS

```
print(STOCH,DET)
print(DET)
print(np.sum(np.array(DET>STOCH),axis=0),np.sum(np.array(STOCH>0),
    axis=0))
```

These last lines are meant to store the performances of both deterministic and stochastic algorithms over the same initial instance

Note that the third line is necessary to print how many times the deterministic algo wins over the stochastic and to say how many non feasible dataset there have been.

Next tables will show you the results on different instances (reading mxn means m tiles with length n)

	Stoch	**Draw**	**Det**
0.5	0.25	0.5	0.25
0.3	0.36	0.29	0.35
0.1	0.45	0.02	0.47

Table 1: Results on a 70x50

	Stoch	**Draw**	**Det**
0.5	0.25	0.5	0.25
0.3	0.35	0.3	0.35
0.1	0.48	0.03	0.49

Table 2: Results on a 30x20

	Stoch	**Draw**	**Det**
0.5	0.3	0.4	0.3
0.3	0.36	0.25	0.38
0.1	0.42	0.08	0.5

Table 3: Results on a 150x200

We can't appreciate any major advantage deriving from using an algorithm instead of another one.

FUTURE IMPROVEMENTS AND IDEAS

- **Computation complexity**

  To reduce the time of execution (particularly high when treating sparse tiles), beam search could be applied to both the algorithms presented here and see whether one improves more than another

- **Optimization of stochastic algo**

  The softmax operator does not improve the deterministic algorithm,so an idea could be to try and find another transformation for the ranks in order to prioritize more efficiently the candidates. Also an adaptive sequence could be implemented.