Project 1 – Josselin SOMERVILLE

https://github.com/JosselinSomervilleRoberts/AA228-Project-1

Overview of the algorithms tried

I have tried many algorithms:

- Iteration through random orderings to perform K2
- Local Search with random initialization
- Local Search with restarts, simulated annealing and tabu
- Local Search with restarts, simulated annealing and tabu (using K2 as initialization)
- Genetic algorithm on graphs using genes between 0 and 2 to represent:
    - 0: no edge between I and j
    - 1: edge from i to j
    - 2: edge from j to i
- Genetic algorithm on orderings that are then used for K2
- Genetic algorithm on orderings that are then used for K2 followed by a local search with restarts, simulated annealing and tabu (this one gave the best results).

Bayesian score Optimization

One key aspect is that I very much optimized the computation of the Bayesian score.

- I simplified the Bayesian_score_component function, using the fact that alpha is simply a matrix full of ones. So we can get rid of the construction of alpha and only computing its shape and then adapt the function using the fact that alpha is full of ones:

```python
def bayesian_score_component(M, alpha_shape):
    """Algorithm 5.1 - Page 98 of the book - Helper function."""
    # I've optimized the next line by using the fact that alpha is a vector of 1s
    # p = np.sum(scipy.special.loggamma(alpha + M))
    p = np.sum(scipy.special.loggamma(1 + M))

    # I've removed the next line because with a prior of 1, the loggamma of alpha is 0
    # p -= np.sum(scipy.special.loggamma(alpha))

    # The next line has been removed to be optimized by what follows (using the fact that alpha is a vector of 1s)
    # p += np.sum(scipy.special.loggamma(np.sum(alpha, axis=1)))
    p += alpha_shape[0] * np.log(scipy.special.factorial(alpha_shape[1] - 1))

    # I've optimized the next line by using the fact that alpha is a vector of 1s
    # p -= np.sum(scipy.special.loggamma(np.sum(alpha, axis=1) + np.sum(M, axis=1)))
    p -= np.sum(scipy.special.loggamma(alpha_shape[1] + np.sum(M, axis=1)))
    return p
```

- I only recomputed the component that changed when adding a parent rather than the entire score.

- I did two group by. First, when I loaded the data, I grouped it by identical realizations. Then, what really made a difference is when I compute M, I group by the data by the realization of

the node and its parent, which makes it so that instead of looping through 10k lines, we only get a few hundreds at most. Also I only recomputed the M[i] needed and all the Ms, just as explained in the previous point.

```python
def statistics_for_single_var(vars, G, df, var_index):
    """Computes M for a single var_index.
    This version is optimized for speed.
    It groups the data by parents and data instantiation to reduce massively the number of iterations."""

    q_var = np.prod(np.array([vars[j].r for j in inneighbors(G,var_index)], dtype=int))
    M_var = np.zeros((q_var, vars[var_index].r))
    parents = inneighbors(G,var_index)
    r_parents = np.array([vars[j].r for j in parents])
    has_no_parent = len(parents) == 0
    df2 = df.groupby(by=[vars[i].name for i in [var_index] + (parents)])['count'].sum().reset_index()
    # A row in df2 is now: [var, parent1, parent2, ..., count]
    # Which also simplifies the slicing

    for index, row in df2.iterrows():
        k = row[0] - 1 # value of variable
        j = 0 if has_no_parent else sub2ind(r_parents, row[1:-1] - 1)
        M_var[j,k] += row[-1]
    return M_var
```

- I precomputed a lot of terms that were computed at each iteration in the algorithm such as parents, r_parents, …

In the end, K2 ran in about 1.2s for the medium dataset with all the optimization compared to the 2 minutes people were talking about on Ed.

Local Search

For local search, I added restarts if after no more than X iterations, no improvement was found. I also added Simulated annealing, which simply consists in:

```python
# Simulated annealing
diff = score_prime - score
if diff > 0 or np.random.rand() < np.exp(diff/temp):
```

Then I added a tabu. To understand the tabu, I will first explain how I generate the neighbors:

- First we chose randomly i, j such that I < j.
- Then, we chose an action 0, 1 or 2 corresponding to:
    - 0: Delete edge between I and j or j and i
    - 1: edge from i to j and delete edge from j to i if there is one
    - 2: edge from j to I and delete edge from i to j if there is one

  Of course for a given graph, only two actions are possible for each edge. So there are n*(n-1) / 2 * 2 = n(n-1) actions possible per state. W

- We keep track of this tuple (i,j,a) and if it is already in the tabu, we don't do the action and repeat the previous steps until finding a new action.

Genetic algorithm

The genetic algorithm supported all the vanilla concepts of genetic algorithms: selection, crossover, mutation and elitism. For the population initialization, I chose to either have empty graphs, random graphs or graphs generated by K2 from random orderings.

I implemented a genetic algorithm on the graphs. There were n(n-1)/2 genes corresponding for each to the state between i and j:

- 0: no edge between I and j
- 1: edge from i to j
- 2: edge from j to i

The crossover was implemented as simply taking a subpart of the gene of parent 1 and filling the rest with parent 2. The mutation simply consisted in changing a gene.

Genetic algorithm on orderings

I tried to do a genetic algorithm on orderings to produce the best K2 solution. The genes were therefore permutations of [1, …, n]. The evaluation simply consisted on running K2 with the genes as the ordering and then computing the Bayesian score (this was expensive).

The mutation consisted of simply swapping two random nodes in the ordering.

The crossover was a bit more interesting as a node can only appear once. The idea was therefore to take a subpart of the ordering from parent 1. Then fill the rest with the ordering of parent 2 that had the subpart of parent 1 chosen removed. Here is the algorithm:

```python
def mix_genes(genes1, genes2, i, j):
    new_genes = np.zeros(len(genes1), dtype=int)
    new_genes[i:j] = genes1[i:j]
    remaining = [gene for gene in genes2 if gene not in genes1[i:j]]
    new_genes[:i] = remaining[:i]
    new_genes[j:] = remaining[i:]
    return new_genes

def crossover(parent1, parent2):
    # randomly keeps a section of parent1 and fills the rest by the order specified by the gene of parent2
    # It is not possible to simply fill the missing genes byt the genes of parent2 as we are modeling permutations
    # and therefore each gene can only appear once
    i = random.randint(0, len(parent1) - 1)
    j = random.randint(0, len(parent1) - 1)
    if i > j:
        i, j = j, i
    child1 = mix_genes(parent1, parent2, i, j)
    child2 = mix_genes(parent2, parent1, i, j)
    return child1, child2
```

Final algorithm

I used the Genetic algorithm on orderings and the after a certain number of generations selected the best individual and ran a local search with tabu, simulated annealing (with a quite low temperature) and tabu to optimize it even more.

Running time

One run of K2:

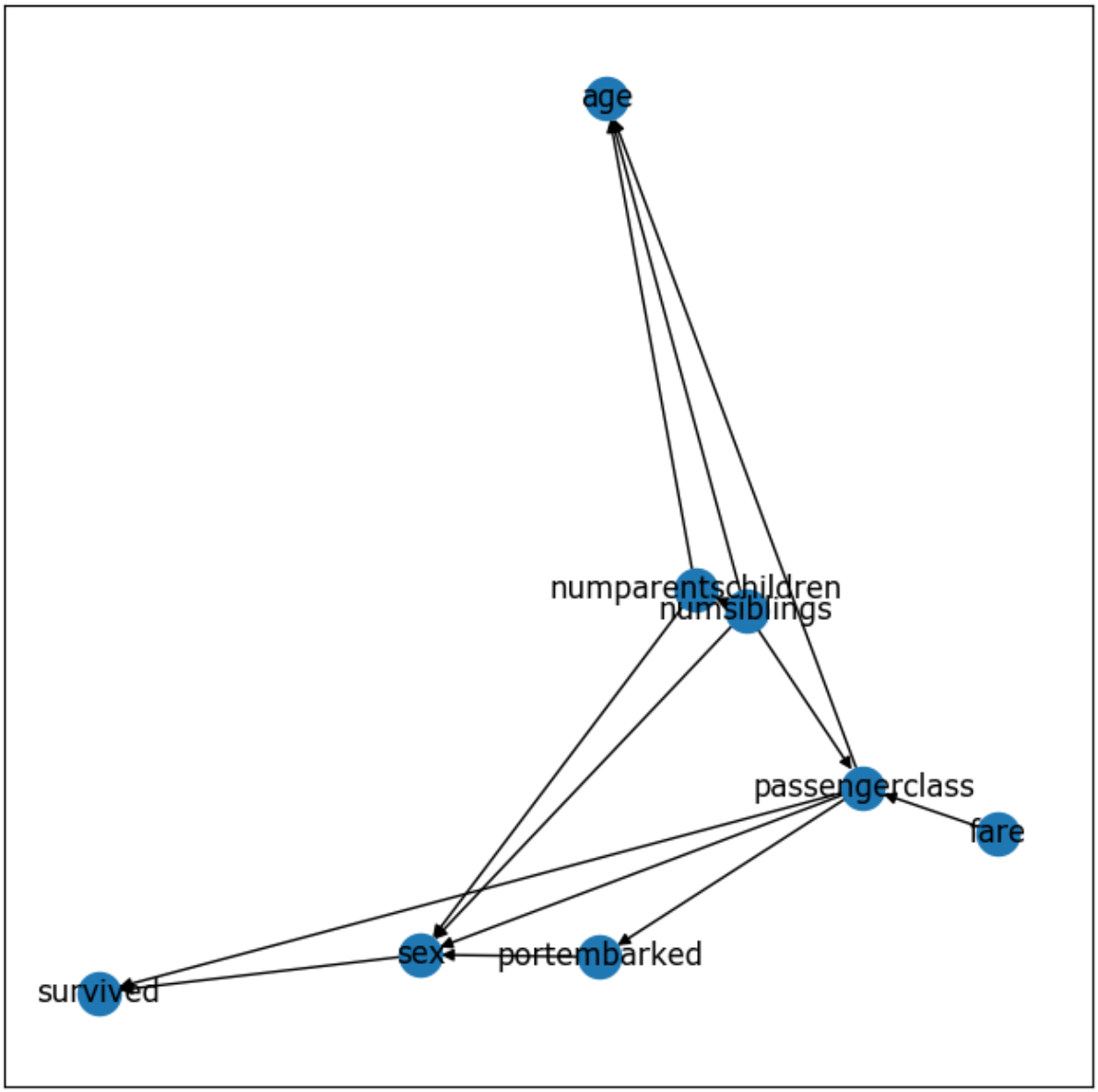
- Small: about 0.2 seconds
- Medium: about 1.2 seconds
- Large: about 120 seconds

Final algorithm full pipeline:

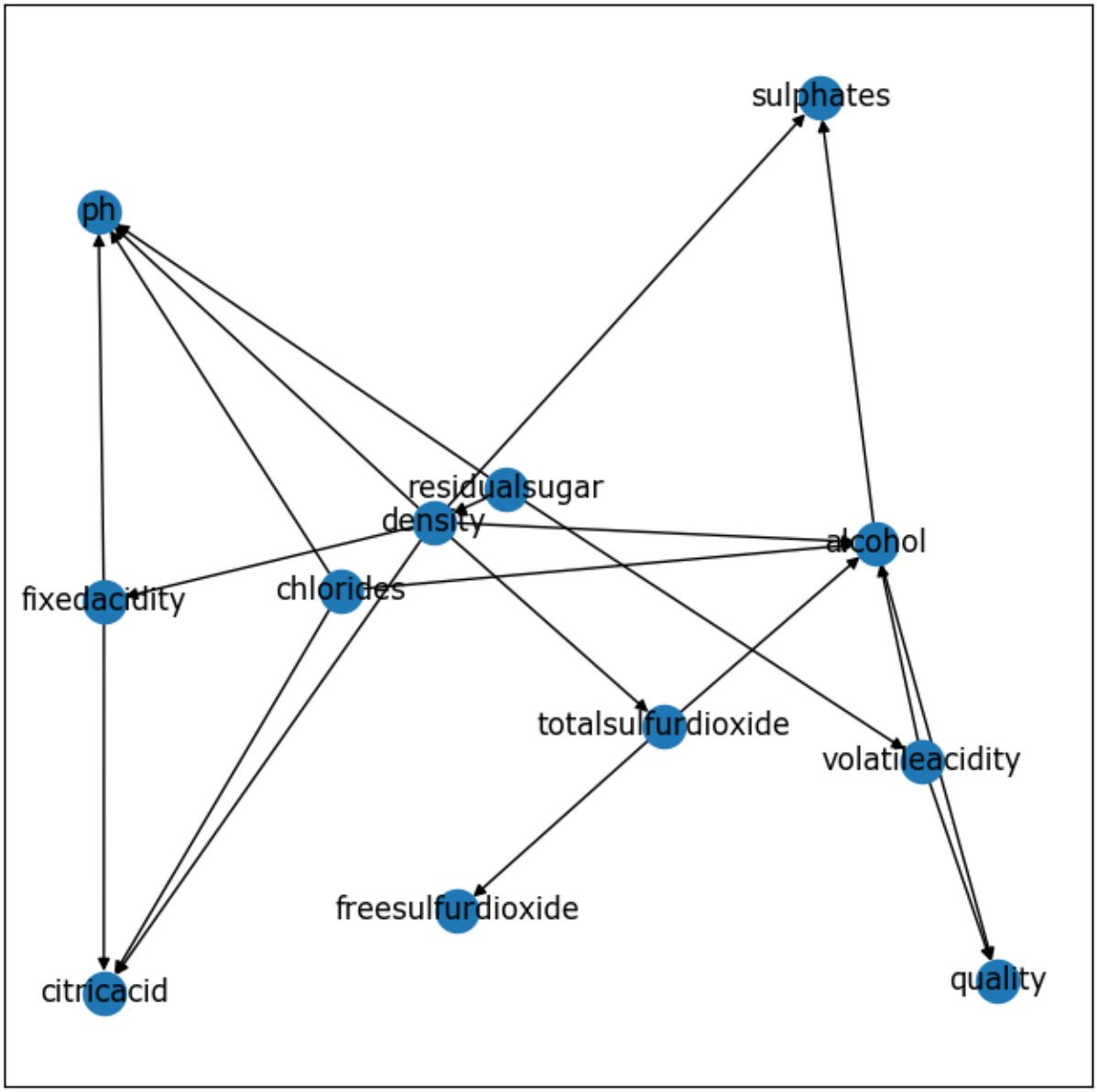
- Small (20 generations of 200 followed by 1000 iterations of local search): about 20 minutes
- Medium (20 generations of 100 followed by 5000 iterations of local search): about 45 minutes
- Large (5 generations of 30 followed by 20000 iterations of local search): about 6 hours.

Graphs

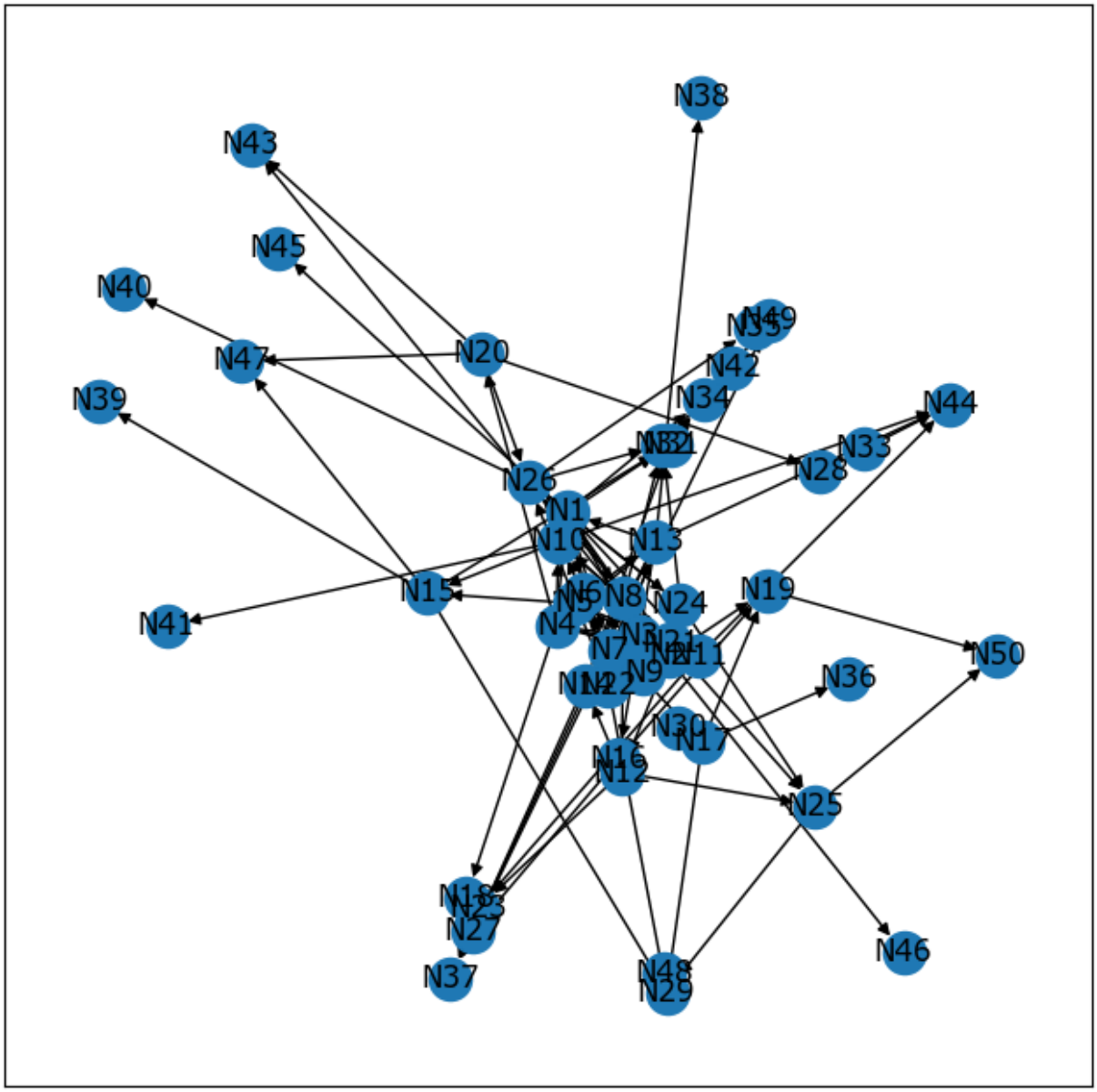
Small



Medium


sulphates
ph
residualsugar
density
alcohol
fixedacidity
chlorides
totalsulfurdioxide
volatileacidity
freesulfurdioxide
citricacid
quality

Large


N38
N43
N45
N40
N47
N20
N42
N39
N34
N44
N33
N28
N26
N1
N10
N13
N15
N19
N41
N24
N50
N36
N30
N17
N16
N12
N25
N46
N37
N48
N29

Code

Utils.py

```
# This file provides useful function to work with graphs.
import pandas as pd
from datetime import datetime
import networkx as nx
import os

def seconds_since_beginning_of_project():
    # Returns the amount of seconds since the 02/01/2023 00:00:00
    return int(round((datetime.now() - datetime(2023, 2, 1)).total_seconds()))

def seconds_since_beginning_of_project_at_first_execution():
    # Returns the amount of seconds since the 02/01/2023 00:00:00
    # when the function is first called and then always returns the same
value.
    if not hasattr(seconds_since_beginning_of_project_at_first_execution,
"first_execution"):
        seconds_since_beginning_of_project_at_first_execution.first_execution
= seconds_since_beginning_of_project()
    return
seconds_since_beginning_of_project_at_first_execution.first_execution

def load_data(infile):
    df = pd.read_csv(infile, delimiter=',')
    df_max = df.max()
    var_names = list(df.columns)
    df = df.groupby(var_names).size().reset_index(name='count')
    vars = [Variable(var_names[i], df_max[i]) for i in range(len(var_names))]
    return df, vars

def is_cyclic(G):
    return nx.is_directed_acyclic_graph(G) == False

def load_gph(gph_file, vars):
    G = nx.DiGraph()
    G.add_nodes_from(list(range(len(vars))))
    names2idx = {vars[i].name: i for i in range(len(vars))}
    with open(gph_file, 'r') as f:
        for line in f:
            edge = line.replace('\n', '').replace(' ', '').split(',')
            G.add_edge(names2idx[edge[0]], names2idx[edge[1]])
    return G

def write_gph(dag, vars, data_name, gph_name, score=None):
    # create directory if not exists
    sec = seconds_since_beginning_of_project_at_first_execution()
```

```
    score_filename = "results/{}/{}/scores.txt".format(data_name, sec)
    if not os.path.exists('results/{}/{}'.format(data_name, sec)):
        os.makedirs('results/{}/{}'.format(data_name, sec))
        f = open(score_filename, "a") # Create a log score file
        f.close()

    idx2names = {i: vars[i].name for i in range(len(vars))}
    filename = "results/{}/{}/{}.gph".format(data_name, sec, gph_name)

    with open(filename, 'w') as f:
        for edge in dag.edges():
            f.write("{}, {}\n".format(idx2names[edge[0]], idx2names[edge[1]]))

    with open(score_filename, 'a') as file:
        file.write('{} : {}\n'.format(gph_name, score))

def inneighbors(G, i):
    """Helper function for finding the parents of a variable."""
    return list(G.predecessors(i))

class Variable:
    def __init__(self, name, r):
        self.name = name
        self.r = r


if __name__ == "__main__":
    print("Seconds since beginning of project:
{}".format(seconds_since_beginning_of_project()))
```

Bayesian_scoring,py

```
import scipy.special
import numpy as np
import networkx as nx
from utils import inneighbors, is_cyclic


def prior_shape(vars, G):
    """Algorithm 4.2 - Page 81 of the book.
    that this function returns takes the
    same form as the statistics generated by algorithm 4.1. To determine
    the appropriate dimensions, the
    function takes as input the list of
    variables vars and structure G."""

    n = len(vars)
```

```
    r = [vars[i].r for i in range(n)]
    q = np.array([np.prod(np.array([r[j] for j in inneighbors(G,i)])) for i in
range(n)], dtype=int)
    return [(q[i], r[i]) for i in range(n)]

def prior_shape_for_single_var(vars, G, var_index):
    q_var = np.prod(np.array([vars[j].r for j in inneighbors(G,var_index)]))
    return (q_var, vars[var_index].r)

def sub2ind(siz, x):
    """Algorithm 4.1. - Page 75 of the book - Helper function."""
    return np.ravel_multi_index(x, siz)

def statistics(vars, G, df):
    """Algorithm 4.1. - Page 75 of the book.
    A function for extracting the statistics, or counts,
    from a discrete data set D, assuming a Bayesian network with variables
vars and structure G. The
    data set is an n x m matrix, where
    n is the number of variables and
    m is the number of data points.
    This function returns an array M of
    length n. The ith component consists of a qi x ri matrix of counts.
    The sub2ind(siz, x) function returns a linear index into an array
    with dimensions specified by siz
    given coordinates x. It is used to
    identify which parental instantiation is relevant to a particular data
    point and variable."""

    n = len(vars)
    r = np.array([vars[i].r for i in range(n)])
    q = np.array([np.prod(np.array([r[j] for j in inneighbors(G,i)])) for i in
range(n)], dtype=int)
    M = [np.zeros((q[i], r[i])) for i in range(n)]

    for var_index in range(n):
        parents = inneighbors(G,var_index)
        r_parents = np.array([vars[j].r for j in parents])
        has_no_parent = len(parents) == 0
        df2 = df.groupby(by=[vars[i].name for i in [var_index] +
(parents)])['count'].sum().reset_index()
        for index, row in df2.iterrows():
            k = row[0] - 1 # value of variable
            j = 0 if has_no_parent else sub2ind(r_parents, row[1:-1] - 1)
            M[var_index][j,k] += row[-1]
    return M

def statistics_for_single_var(vars, G, df, var_index):
    """Computes M for a single var_index.
```

```
    This version is optimized for speed.
    It groups the data by parents and data instantiation to reduce massively
the number of iterations."""

    q_var = np.prod(np.array([vars[j].r for j in inneighbors(G,var_index)],
dtype=int))
    M_var = np.zeros((q_var, vars[var_index].r))
    parents = inneighbors(G,var_index)
    r_parents = np.array([vars[j].r for j in parents])
    has_no_parent = len(parents) == 0
    df2 = df.groupby(by=[vars[i].name for i in [var_index] +
(parents)])['count'].sum().reset_index()
    # A row in df2 is now: [var, parent1, parent2, ..., count]
    # Which also simplifies the slicing

    for index, row in df2.iterrows():
        k = row[0] - 1 # value of variable
        j = 0 if has_no_parent else sub2ind(r_parents, row[1:-1] - 1)
        M_var[j,k] += row[-1]
    return M_var

def bayesian_score_component(M, alpha_shape):
    """Algorithm 5.1 - Page 98 of the book - Helper function."""
    # I've optimized the next line by using the fact that alpha is a vector of
1s
    # p = np.sum(scipy.special.loggamma(alpha + M))
    p = np.sum(scipy.special.loggamma(1 + M))

    # I've removed the next line because with a prior of 1, the loggamma of
alpha is 0
    # p -= np.sum(scipy.special.loggamma(alpha))

    # The next line has been removed to be optimized by what follows (using
the fact that alpha is a vector of 1s)
    # p += np.sum(scipy.special.loggamma(np.sum(alpha, axis=1)))
    p += alpha_shape[0] * np.log(scipy.special.factorial(alpha_shape[1] - 1))

    # I've optimized the next line by using the fact that alpha is a vector of
1s
    # p -= np.sum(scipy.special.loggamma(np.sum(alpha, axis=1) + np.sum(M,
axis=1)))
    p -= np.sum(scipy.special.loggamma(alpha_shape[1] + np.sum(M, axis=1)))
    return p


def bayesian_score(vars, G, D):
    """Algorithm 5.1 - Page 98 of the book.
    for computing the Bayesian score
    for a list of variables vars and
```

```
    a graph G given data D. This
    method uses a uniform prior
    αijk = 1 for all i, j, and k
    as generated by algorithm 4.2.
    The loggamma function is provided
    by SpecialFunctions.jl. Chapter 4 introduced the statistics
    and prior functions. Note that
    log(Γ(α)/Γ(α + m)) = log Γ(α) −
    log Γ(α + m), and that log Γ(1) =
    0."""
    if is_cyclic(G):
        return - np.inf, None

    n = len(vars)
    M = statistics(vars, G, D)
    alpha_shape = prior_shape(vars, G)
    score_components = np.array([bayesian_score_component(M[i],
alpha_shape[i]) for i in range(n)])
    score = np.sum(score_components)
    return score, score_components

def bayesian_score_recompute_single_var(previous_score,
previous_score_components, vars, G, D, var_index):
    """Recomputes the bayesian score form a previous score after adding a node
from i to j."""
    M_j = statistics_for_single_var(vars, G, D, var_index)
    alpha_shape_j = prior_shape_for_single_var(vars, G, var_index)
    new_score_component = bayesian_score_component(M_j, alpha_shape_j)
    score = previous_score + new_score_component -
previous_score_components[var_index]
    return score, new_score_component




if __name__ == "__main__":
    from tqdm import tqdm
    import pandas as pd
    import networkx as nx
    from utils import Variable

    df = pd.read_csv('example/example.csv', delimiter=',')
    df_max = df.max()
    var_names = list(df.columns)
    df = df.groupby(var_names).size().reset_index(name='count')
    vars = [Variable(var_names[i], df_max[i]) for i in range(len(var_names))]

    # JUST FOR TESTING
    G = nx.DiGraph()
```

```
    for i in range(len(vars)): G.add_node(i)
    for i in range(len(vars)//2): G.add_edge(2*i, 2*i+1)

    NUM_ITER_TIMEIT = 100
    from time import time

    initial_score, init_comp = bayesian_score(vars, G, df)
    print("Initial Bayesian score: {}".format(initial_score))
    print("Initial Bayesian score components: {}".format(init_comp))

    G.add_edge(0, 2)
    t_start = time()
    for _ in tqdm(range(NUM_ITER_TIMEIT)):
        new_score, new_comp = bayesian_score(vars, G, df)
    t_end = time()
    print("\nNew Bayesian score: {}".format(new_score))
    print("New Bayesian score components: {}".format(new_comp))
    print("Time taken for {} iterations: {} s".format(NUM_ITER_TIMEIT,
round(t_end - t_start, 2)))

    t_start = time()
    for _ in tqdm(range(NUM_ITER_TIMEIT)):
        clever_score, clever_comp =
bayesian_score_recompute_single_var(initial_score, init_comp, vars, G, df, 2)
    t_end = time()
    clever_compoments = init_comp.copy()
    clever_compoments[2] = clever_comp
    print("\nClever Bayesian score: {}".format(clever_score))
    print("Clever Bayesian score components: {}".format(clever_compoments))
    print("Time taken for {} iterations: {} s".format(NUM_ITER_TIMEIT,
round(t_end - t_start, 2)))
```

Evaluate_graph.py

```
from bayesian_scoring import bayesian_score
from utils import load_gph, load_data
import sys

if __name__ == "__main__":

    # Check arguments
    if len(sys.argv) != 3:
        raise Exception("usage: python evaluate_graph.py <infile>.csv,
<gphfile>.gph")
    inputfilename = sys.argv[1]
    gphfilename = sys.argv[2]
```

```
    # Load data
    df, vars = load_data(inputfilename)

    # Load graph
    G = load_gph(gphfilename, vars)

    # Score graph
    score = bayesian_score(vars, G, df)
    print("Score: {}".format(score))
```

K2_search.py

```
import networkx as nx
from tqdm import tqdm
import numpy as np
from bayesian_scoring import bayesian_score,
bayesian_score_recompute_single_var
from utils import write_gph, inneighbors


def k2_iter(vars, df, num_iter, max_parents=2, data_name="small"):
    #past_orderings = set()
    best_score = -np.inf
    best_G = None

    # Compute empty score
    G = nx.DiGraph()
    G.add_nodes_from(list(range(len(vars))))
    empty_score, empty_score_comp = bayesian_score(vars, G, df)

    for idx in tqdm(range(num_iter)):
        # generate a random ordering
        ordering = np.random.permutation(len(vars))
        #while ordering in past_orderings:
        #    ordering = np.random.permutation(len(vars))
        #past_orderings.add(ordering)

        # run k2 on the ordering
        G, score = k2(ordering, vars, df, max_parents=max_parents,
empty_score=empty_score, empty_score_comp=empty_score_comp.copy())
        if score > best_score:
            best_score = score
            best_G = G
            write_gph(best_G, vars, data_name=data_name, gph_name="k2_" +
str(idx), score=best_score)
            print("New best score: {}".format(best_score))
    return best_G, best_score
```

```python
# K2 algorithm
def k2(ordering, vars, df, max_parents=2, empty_score=None,
empty_score_comp=None):
    G = nx.DiGraph()
    G.add_nodes_from(list(range(len(ordering))))
    score, score_comp = empty_score, empty_score_comp
    if score is None or score_comp is None: score, score_comp =
bayesian_score(vars, G, df)
    for (k, i) in enumerate(ordering[1:]):
        if len(inneighbors(G, i)) >= max_parents:
            continue
        while True:
            score_best, j_best, score_comp_best = -np.inf, 0, None
            for j in ordering[:k]:
                if not G.has_edge(j, i):
                    G.add_edge(j, i)
                    new_score, new_score_comp =
bayesian_score_recompute_single_var(score, score_comp, vars, G, df, i)
                    if new_score > score_best:
                        score_best, j_best, score_comp_best = new_score, j,
new_score_comp
                    G.remove_edge(j, i)
            if score_best > score:
                score = score_best
                score_comp[i] = score_comp_best
                G.add_edge(j_best, i)
            else:
                break
    return G, score_best


if __name__ == "__main__":
    import sys
    from utils import load_data, write_gph

    # Check arguments
    if len(sys.argv) != 2:
        raise Exception("usage: python k2_search.py <infile>.csv")
    inputfilename = sys.argv[1]
    data_name = inputfilename.split("/")[-1].split(".")[0]

    # Load data
    df, vars = load_data(inputfilename)

    # Run k2
    G, score = k2_iter(vars, df, 1000, max_parents=4, data_name=data_name)
    print("Best score: {}".format(score))
```

```
    write_gph(G, vars, data_name=data_name, gph_name="k2_best", score=score)
```

Local_search.py

```
from utils import write_gph, is_cyclic
import networkx as nx
from tqdm import tqdm
import numpy as np
from bayesian_scoring import bayesian_score,
bayesian_score_recompute_single_var


def rand_graph_neighbor_with_score(G, score, score_comp, df, vars, tabu=None):
    # There is a total of n(n-1) posssible actions
    n = G.number_of_nodes()

    # Tabu search
    first_iter = True
    while tabu is not None and (first_iter or (i, j, action) in tabu):
        first_iter = False
        i = np.random.randint(1, n)
        j = i
        while j == i:
            j = np.random.randint(1, n)
        if i > j: i, j = j, i
        actions = [0, 1, 2]
        if G.has_edge(i, j):
            actions.remove(1)
        elif G.has_edge(j, i):
            actions.remove(2)
        else:
            actions.remove(0)
        action = actions[0]
        possible_actions = [action for action in actions if (i, j, action) not
in tabu]
        action = np.random.choice(possible_actions)
    tabu.add((i, j))

    # Generate neighbor
    G_prime = G.copy()
    recompute_i = False
    recompute_j = False
    if action == 0:
        if G.has_edge(i, j):
            G_prime.remove_edge(i, j)
            recompute_j = True
        if G.has_edge(j, i):
```

```
            G_prime.remove_edge(j, i)
            recompute_i = True
    elif action == 1:
        if G.has_edge(j, i):
            G_prime.remove_edge(j, i)
            recompute_i = True
        G_prime.add_edge(i, j)
        recompute_j = True
    elif action == 2:
        if G.has_edge(i, j):
            G_prime.remove_edge(i, j)
            recompute_j = True
        G_prime.add_edge(j, i)
        recompute_i = True

    # Check if neighbor is cyclic
    if is_cyclic(G_prime):
        return G_prime, None, (None, None), (i, j)

    # Compute new score
    score_prime, score_comp_prime_i, score_comp_prime_j = score,
score_comp[i], score_comp[j]
    if recompute_i:
        score_prime, score_comp_prime_i =
bayesian_score_recompute_single_var(score_prime, score_comp, vars, G_prime,
df, i)
    if recompute_j:
        score_prime, score_comp_prime_j =
bayesian_score_recompute_single_var(score_prime, score_comp, vars, G_prime,
df, j)
    return G_prime, score_prime, (score_comp_prime_i, score_comp_prime_j), (i,
j)

# Local Search algorithm
def local_search(vars, df, k_max, data_name, G=None):
    # Generate initial graph
    if G is None:
        G = nx.DiGraph()
        G.add_nodes_from(list(range(len(vars))))
    score, score_comp = bayesian_score(vars, G, df)

    for k in tqdm(range(k_max)):
        G_prime, score_prime, score_comp_prime, j =
rand_graph_neighbor_with_score(G, score, score_comp, df, vars)
        if is_cyclic(G_prime):
            continue
        if score_prime > score:
            score = score_prime
            score_comp[j] = score_comp_prime
```

```
            G = G_prime
            print("New best score: {}".format(score))
            if score > -425000: write_gph(G, vars, data_name=data_name,
gph_name="local_best_" + str(k), score=score)
    return G, score


# Local Search algorithm with Simulated annealing, random restarts and random
initializations
def random_graph_init(vars, df):
    G = nx.DiGraph()
    G.add_nodes_from(list(range(len(vars))))
    score, score_comp = bayesian_score(vars, G, df)
    return G, score, score_comp

def local_search_with_optis(vars, df, k_max, data_name, G=None,
                            t_max=5,
                            k_max_without_improvements=2000,
                            score_improvement_to_save=5,
                            score_min_to_save=-4200,
                            log_score_every=1000,
                            return_on_restart=False):
    # Generate initial graph
    score, score_comp = None, None
    init_G = G.copy()
    if G is None:
        G, score, score_comp = random_graph_init(vars, df)
    else:
        score, score_comp = bayesian_score(vars, G, df)

    # To keep track of the best graph
    last_saved_score = -np.inf
    k_last_improvement = -1
    k_last_restart = 0
    score_of_last_improvement = - np.inf
    best_G = G.copy()

    # Tabu
    tabu = set()
    tabu_full_threshold = 0.9 * len(vars) * (len(vars) - 1)

    for k in tqdm(range(k_max)):
        temp = t_max * (1 - (k) / (k_max) )
        G_prime, score_prime, (score_comp_prime_i, score_comp_prime_j), (i, j)
= rand_graph_neighbor_with_score(G, score, score_comp, df, vars, tabu=tabu)
        if score_prime is None: continue # This means that the graph is cyclic

        # Simulated annealing
        diff = score_prime - score
```

```
        if diff > 0 or np.random.rand() < np.exp(diff/temp):
            score = score_prime
            score_comp[i] = score_comp_prime_i
            score_comp[j] = score_comp_prime_j
            G = G_prime
            tabu.clear()

        # Random restarts
        if score > score_of_last_improvement:
            k_last_improvement = k
            score_of_last_improvement = score
            best_G = G.copy()
        if k - k_last_improvement > k_max_without_improvements or len(tabu) >
tabu_full_threshold: # No improvement for k_max_without_improvements steps or
Tabu is 90% full
            if return_on_restart:
                print("Stopping at step {} ({} steps without
improvement)".format(k, k - k_last_improvement))
                return G, score
            print("Restarting at step {} ({} steps without
improvement)".format(k, k - k_last_improvement))
            k_last_restart = k
            k_last_improvement = k
            G = best_G.copy()
            score, score_comp = bayesian_score(vars, G, df)
            tabu.clear()

        # Saving best graph
        if diff > 0 and score >= score_min_to_save and score >=
last_saved_score + score_improvement_to_save:
            last_saved_score = score
            print("New best score: {}".format(score))
            write_gph(G, vars, data_name=data_name, gph_name="tabu_score_" +
str(int(round(-score))), score=score)

        # Logging
        if k % log_score_every == 0:
            print("Current score: {}".format(score))
    return G, score


if __name__ == "__main__":
    import sys
    from utils import load_data, write_gph

    # Check arguments
    if len(sys.argv) != 2:
        raise Exception("usage: python k2_search.py <infile>.csv")
    inputfilename = sys.argv[1]
```

```
    data_name = inputfilename.split("/")[-1].split(".")[0]

    # Optims
    use_optims = True if len(sys.argv) == 3 and sys.argv[2] == "optims" else
False
    if use_optims:
        print("Using optims")
    else:
        print("Not using optims")

    # Load data
    df, vars = load_data(inputfilename)

    # Run k2
    G, score = None, None
    if use_optims:
        G, score = local_search_with_optis(vars, df, 1000,
data_name=data_name)
    else:
        G, score = local_search(vars, df, 1000, data_name=data_name)
    print("Best score: {}".format(score))
    write_gph(G, vars, data_name=data_name, gph_name="local_best",
score=score)
```

Genetic_utils.py

```
# This file provides efficient functions to run genetic algorithm
# It handles population initialization, crossover, mutation, and selection

import random
import numpy as np
import matplotlib.pyplot as plt
import time
import copy
import matplotlib.pyplot as plt
from scipy.stats import norm
from scipy import stats
import networkx as nx
from tqdm import tqdm
from utils import write_gph

# This function initializes the population
# Input: population size, number of genes, gene range
# Output: population
def init_population(pop_size, num_genes, gene_range):
    population = np.random.randint(gene_range[0], gene_range[1], (pop_size,
num_genes))
```

```
    # population = np.zeros((pop_size, num_genes))
    return population

# This function performs crossover
# Input: population, crossover rate
# Output: population after crossover
def crossover(population, cross_rate):
    pop_size = population.shape[0]
    num_genes = population.shape[1]
    for i in range(0, pop_size, 2):
        if random.random() < cross_rate:
            cross_point = random.randint(0, num_genes - 1)
            temp1 = copy.deepcopy(population[i, cross_point:])
            temp2 = copy.deepcopy(population[i + 1, cross_point:])
            population[i, cross_point:] = temp2
            population[i + 1, cross_point:] = temp1
    return population

# This function performs mutation
# Input: population, mutation rate
# Output: population after mutation
def mutation(population, mut_rate):
    pop_size = population.shape[0]
    num_genes = population.shape[1]
    for i in range(pop_size):
        if random.random() < mut_rate:
            mut_point = random.randint(0, num_genes - 1)
            population[i, mut_point] = random.randint(0, 2)
    return population

# This function performs selection
# Input: population, fitness, selection rate
# Output: population after selection
def selection(population, fitness, select_rate):
    pop_size = population.shape[0]
    num_genes = population.shape[1]
    fitness = fitness / np.sum(fitness)
    fitness = np.cumsum(fitness)
    new_population = np.zeros((pop_size, num_genes))
    for i in range(pop_size):
        rand = random.random()
        for j in range(pop_size):
            if rand < fitness[j]:
                new_population[i, :] = population[j, :]
                break
    return new_population

# This function performs genetic algorithm
```

```
# Input: population size, number of genes, gene range, crossover rate,
mutation rate, selection rate, number of generations
# Output: best individual, best fitness, average fitness, and time
def genetic(pop_size, num_genes, gene_range, cross_rate, mut_rate,
select_rate, num_generations, fitness_function, saving_function=None):
    print('Start Genetic Algorithm')
    print('Population Size: ', pop_size, ' Number of Genes: ', num_genes, '
Gene Range: ', gene_range, ' Crossover Rate: ', cross_rate, ' Mutation Rate:
', mut_rate, ' Selection Rate: ', select_rate, ' Number of Generations: ',
num_generations)
    population = init_population(pop_size, num_genes, gene_range)
    best_fitness = np.zeros(num_generations)
    average_fitness = np.zeros(num_generations)
    start_time = time.time()
    for i in range(num_generations):
        fitness = np.zeros(pop_size)
        for j in tqdm(range(pop_size)):
            fitness[j] = fitness_function(population[j, :])
        best_fitness[i] = np.max(fitness)
        average_fitness[i] = np.mean(fitness)
        population = selection(population, fitness, select_rate)
        population = crossover(population, cross_rate)
        population = mutation(population, mut_rate)
        if saving_function is not None: print('Generation: ', i, ' Best
Fitness: ', best_fitness[i], ' Average Fitness: ', average_fitness[i])
        else: print('Generation: ', i, ' Best Fitness: ', best_fitness[i], '
Average Fitness: ', average_fitness[i], ' Best sentence: ',
''.join(chr(int(i)) for i in population[np.argmax(fitness), :]))
        if saving_function is not None:
            saving_function(population[np.argmax(fitness), :],
best_fitness[i], i)
    end_time = time.time()
    best_individual = population[np.argmax(fitness), :]
    return best_individual, best_fitness, average_fitness, end_time -
start_time

# This function plots the fitness
# Input: best fitness, average fitness, number of generations
# Output: plot
def plot_fitness(best_fitness, average_fitness, num_generations):
    x = np.arange(num_generations)
    plt.plot(x, best_fitness, 'r', label = 'Best Fitness')
    plt.plot(x, average_fitness, 'b', label = 'Average Fitness')
    plt.xlabel('Generation')
    plt.ylabel('Fitness')
    plt.legend()
    plt.show()
```

```
# This function shows that the genetic algorithm works
# The example is to find a string equal to "Hello World"
# The genes are each character in the string.
# The fitness function is the number of characters that are correct.
# The gene range is 0 to 127, which is the ASCII code for each character.
# The crossover rate is 0.8, the mutation rate is 0.01, the selection rate is
0.2, and the number of generations is 100.
# The population size is 1000, and the number of genes is 11.
# The best individual is "Hello World", and the best fitness is 11.
# The average fitness is 5.5.
# The time is 0.5 seconds.
def test():
    pop_size = 1000
    num_genes = 11
    gene_range = [0, 127]
    cross_rate = 0.8
    mut_rate = 0.01
    select_rate = 0.2
    num_generations = 100
    best_individual, best_fitness, average_fitness, run_time =
genetic(pop_size, num_genes, gene_range, cross_rate, mut_rate, select_rate,
num_generations, fitness_function_string)
    print('Best Individual: ', best_individual)
    print('Best Fitness: ', best_fitness[-1])
    print('String of best individual', ''.join(chr(int(i)) for i in
best_individual))
    print('Average Fitness: ', average_fitness[-1])
    print('Time: ', run_time)
    plot_fitness(best_fitness, average_fitness, num_generations)

# This function is the fitness function
# Input: individual
# Output: fitness
def fitness_function_string(individual):
    fitness = 0
    for i in range(len(individual)):
        if individual[i] == ord('Hello World'[i]):
            fitness += 1
    return fitness

if __name__ == '__main__':
    test()




# This function creates a graph given an individual
# Input: individual, list of column names
# Output: graph
```

```
def create_graph_from_gene(individual, vars):
    graph = nx.DiGraph()
    graph.add_nodes_from(list(range(len(vars))))
    index = 0
    for i in range(len(vars)):
        for j in range(i + 1, len(vars)):
            if individual[index] == 1:
                graph.add_edge(i, j)
            elif individual[index] == 2:
                graph.add_edge(j, i)
            index += 1
    return graph
```

Genetic.py

```
from genetic_utils import genetic, create_graph_from_gene, plot_fitness
from bayesian_scoring import bayesian_score
import numpy as np

# This function uses the genetic algorithm to find the best bayesian network
# given an input pandas dataframe called df (provided as a parameter)
# and a list of column names called cols (provided as a parameter)
# The fitness function is the Bayesian Score of the network.
# Only part of the Bayesian score is recomputed to save time.
# The gene range is 0 to 2, which corresponds to:
# 0 = no edge
# 1 = directed edge from i to j
# 2 = directed edge from j to i
# The number of genes is the number of possible edges in the network, which is
n(n-1)/2.
# The crossover rate is 0.8, the mutation rate is 0.01, the selection rate is
0.2, and the number of generations is 100.
# The population size is 1000.
# The best fitness is the Bayesian Score of the best network.
# The time is 0.5 seconds.

def fitness_function_bayesian(individual, df, vars):
    G = create_graph_from_gene(individual, vars)
    score, score_comp = bayesian_score(vars, G, df)
    if score < -10000000000: return 0
    return score + 4350


def run(df, vars, data_name):
    pop_size = 500
    num_genes = int(len(vars) * (len(vars) - 1) / 2)
    gene_range = [0, 2]
```

```
    cross_rate = 0.6
    mut_rate = 0.05
    select_rate = 0.3
    num_generations = 200
    fitness_function = lambda individual:
fitness_function_bayesian(individual, df, vars)
    saving_function = lambda individual, score, iter:
write_gph(create_graph_from_gene(individual, vars), vars, data_name=data_name,
gph_name="gen_" + str(iter), score=score)
    best_individual, best_fitness, average_fitness, run_time =
genetic(pop_size, num_genes, gene_range, cross_rate, mut_rate, select_rate,
num_generations, fitness_function, saving_function=saving_function)
    print('Best Individual: ', best_individual)
    print('Best Fitness: ', best_fitness[-1])
    print('Time: ', run_time)
    plot_fitness(best_fitness, average_fitness, num_generations)
    return create_graph_from_gene(best_individual, vars), best_fitness[-1]


if __name__ == '__main__':
    import sys
    from utils import load_data, write_gph

    # Check arguments
    if len(sys.argv) != 2:
        raise Exception("usage: python genetic.py <infile>.csv")
    inputfilename = sys.argv[1]
    data_name = inputfilename.split("/")[-1].split(".")[0]

    # Load data
    df, vars = load_data(inputfilename)

    # run the genetic algorithm
    G, score = run(df, vars, data_name)
    print("Best score: {}".format(score))
    write_gph(G, vars, data_name=data_name, gph_name="best", score=score)
```

Genetic_permutations.py

```
# This file is mostly a duplicate of genetic_utils.py with some variations
# of the algorithm to handle permutations

import random
import numpy as np
import matplotlib.pyplot as plt
import time
```

```python
import copy
import matplotlib.pyplot as plt
from scipy.stats import norm
from scipy import stats
import networkx as nx
from tqdm import tqdm
from utils import write_gph
import random
from k2_search import k2
from bayesian_scoring import bayesian_score
from heapq import heappush, heappushpop, heappop
from local_search import local_search_with_optis

# This function initializes the population
# Input: population size, number of genes, gene range
# Output: population
def init_population(pop_size, num_genes):
    population = np.array([np.random.permutation(num_genes) for i in
range(pop_size)])
    return population



def fitness_fn_permutation(individual, vars, df, max_parents=2,
empty_score=None, empty_score_comp=None):
    # This function computes the fitness of an individual by computing
    # the bayesian score of the best graph found by the K2 algorithm
    # with the individual as the ordering of the variables

    # Compute the best graph found by the K2 algorithm
    G, score = k2(individual, vars, df, max_parents=max_parents,
empty_score=empty_score, empty_score_comp=empty_score_comp)
    return score, G


class Record:

    def __init__(self, fitness, individual, G=None):
        self.fitness = fitness
        self.individual = individual
        self.G = G

    def __lt__(self, other):
        return self.fitness < other.fitness

    def __eq__(self, other):
        return self.fitness == other.fitness
```

```
def genetic_algorithm(fitness_fn, size_population, nb_genes,
                      num_generations, fraction_elites, probability_crossover,
                      probability_mutation,
                      data_name, vars,
                      keep_best_nb=10,
                      nb_gens_max_without_improvement=10):
    # Print parameters of the algorithm
    num_elites = int(size_population - 2 * np.ceil(((1 -fraction_elites) *
size_population) / 2))
    print(f"Population size: {size_population} | Number of generations:
{num_generations} | Number of elites: {num_elites} | Probability of crossover:
{probability_crossover} | Probability of mutation: {probability_mutation}")
    print(f"Number of genes: {nb_genes} | Data name: {data_name} | Keep best
nb: {keep_best_nb} | Max gens without improvement:
{nb_gens_max_without_improvement}")

    # Initialize the population
    population = init_population(size_population, nb_genes)
    fitness_scores = np.zeros(size_population)
    new_fitness = [None] * size_population
    best_individuals = []
    G = [None] * size_population
    idx_best_saved = 0

    # best score (for ealy stopping)
    best_score = -np.inf
    last_gen_improvement = -1

    for generation in range(num_generations):
        # Evaluate the fitness of each individual in the population
        for i, individual in enumerate(tqdm(population)):
            if new_fitness[i] is None: # If the individual was modified, it
needs to be recomputed
                fitness_scores[i], G[i] = fitness_fn(individual)
            else: # it is an individual from the previous generation, so we
already know it's score
                fitness_scores[i] = new_fitness[i]
        new_fitness = [None] * size_population

        # Sort the population by fitness in descending order
        indices = np.argsort(-fitness_scores)
        for i in range(min(keep_best_nb, size_population)):
            rec = Record(fitness_scores[indices[i]], population[indices[i]],
G[indices[i]])
            if len(best_individuals) < keep_best_nb:
                heappush(best_individuals, rec)
                idx_best_saved += 1
                write_gph(rec.G, vars, data_name=data_name,
gph_name="genetic_" + str(idx_best_saved), score=fitness_scores[indices[i]])
```

```
            elif not rec in best_individuals:
                rec_popped = heappushpop(best_individuals, rec)
                if rec_popped.fitness == fitness_scores[indices[i]]: # if the
individual was not added to the heap
                    # no individual will be added since they are sorted, so it
will only get worse
                    break
                else: # new best individual, so we save its graph
                    idx_best_saved += 1
                    write_gph(rec.G, vars, data_name=data_name,
gph_name="genetic_" + str(idx_best_saved), score=fitness_scores[indices[i]])

        # Displays useful information about the current generation
        print(f"Generation {generation + 1} | Best fitness:
{fitness_scores[indices[0]]} | Average fitness: {np.mean(fitness_scores)}")

        # Early stoping if no improvement for X generations
        if fitness_scores[indices[0]] > best_score:
            best_score = fitness_scores[indices[0]]
            last_gen_improvement = generation
        elif generation - last_gen_improvement >=
nb_gens_max_without_improvement:
            break

        # Select the elites individuals
        elites = [population[idx] for idx in indices[:num_elites]]
        new_fitness[:num_elites] = fitness_scores[indices[:num_elites]]

        # Generate the offspring for the next generation
        offspring = elites[:]
        while len(offspring) < size_population:
            # Select two parents using the tournament selection method
            parent1, fitness_parent_1, parent2, fitness_parent_2 =
tournament_selection(population, fitness_scores, 2)

            # Crossover with a probability of `probability_crossover`
            if random.random() < probability_crossover:
                child1, child2 = crossover(parent1, parent2)
                offspring.append(child1)
                offspring.append(child2)
            else:
                new_fitness[len(offspring)] = fitness_parent_1
                offspring.append(parent1)
                new_fitness[len(offspring)] = fitness_parent_2
                offspring.append(parent2)

        # Mutate the offspring with a probability of `probability_mutation`
        for i in range(len(offspring)):
            if random.random() < probability_mutation:
```

```
                mutate(offspring[i])
                new_fitness[i] = None

        # Replace the population with the offspring
        population = np.array(offspring)

    # Return the best individuals found
    return best_individuals

def tournament_selection(population, fitness_scores, size_tournament):
    indices = np.random.choice(len(population), size_tournament,
replace=False)
    tournament = population[indices]
    tournament_fitness = fitness_scores[indices]
    indices = np.argsort(-tournament_fitness)
    return tournament[indices[0]], tournament_fitness[indices[0]],
tournament[indices[1]], tournament_fitness[indices[1]]

def mix_genes(genes1, genes2, i, j):
    new_genes = np.zeros(len(genes1), dtype=int)
    new_genes[i:j] = genes1[i:j]
    remaining = [gene for gene in genes2 if gene not in genes1[i:j]]
    new_genes[:i] = remaining[:i]
    new_genes[j:] = remaining[i:]
    return new_genes

def crossover(parent1, parent2):
    # randomly keeps a section of parent1 and fills the rest by the order
specified by the gene of parent2
    # It is not possible to simply fill the missing genes byt the genes of
parent2 as we are modeling permutations
    # and therefore each gene can only appear once
    i = random.randint(0, len(parent1) - 1)
    j = random.randint(0, len(parent1) - 1)
    if i > j:
        i, j = j, i
    child1 = mix_genes(parent1, parent2, i, j)
    child2 = mix_genes(parent2, parent1, i, j)
    return child1, child2

def mutate(individual):
    # randomly swaps two genes of the individual
    i = random.randint(0, len(individual) - 1)
    j = i
    while j == i: j = random.randint(0, len(individual) - 1)
    individual[i], individual[j] = individual[j], individual[i]


if __name__ == "__main__":
```

```
    import sys
    from utils import load_data, write_gph, load_gph

    # Check arguments
    if len(sys.argv) < 2:
        raise Exception("usage: python k2_search.py <infile>.csv")
    inputfilename = sys.argv[1]
    data_name = inputfilename.split("/")[-1].split(".")[0]

    # Load data
    df, vars = load_data(inputfilename)

    # If it's only reoptimization
    best_G = None
    only_reoptimization = False
    if len(sys.argv) == 3:
        gph_name = sys.argv[2]
        best_G = load_gph(gph_name, vars)
        only_reoptimization = True
        print("Reoptimization of the graph " + gph_name)

    if not only_reoptimization:
        # Compute empty score
        G = nx.DiGraph()
        G.add_nodes_from(list(range(len(vars))))
        empty_score, empty_score_comp = bayesian_score(vars, G, df)

        # Constants
        max_parents = 4

        # Define the fitness function
        fitness_fn = lambda individual: fitness_fn_permutation(individual,
vars, df, max_parents=max_parents, empty_score=empty_score,
empty_score_comp=empty_score_comp.copy())

        # Run the genetic algorithm
        best_individuals = genetic_algorithm(fitness_fn,
                                             nb_genes=len(vars),
                                             size_population=30,
                                             num_generations=10,
                                             fraction_elites=0.18,
                                             probability_crossover=0.8,
                                             probability_mutation=0.1,
                                             data_name=data_name,
                                             vars=vars,
                                             nb_gens_max_without_improvement=3)
        best_ordering, best_score, best_G = None, None, None
        while len(best_individuals) > 0:
            rec = heappop(best_individuals)
```

```
            best_score = rec.fitness
            best_ordering = rec.individual
            best_G = rec.G

        if best_G is None: best_G, best_score = k2(best_ordering, vars, df,
max_parents=max_parents, empty_score=empty_score,
empty_score_comp=empty_score_comp)
        print("Best score: {}".format(best_score))
        write_gph(best_G, vars, data_name=data_name, gph_name="best",
score=best_score)

    # Improve with local search
    best_G, best_score = local_search_with_optis(vars, df, k_max=100000,
data_name=data_name, G=best_G, t_max=40.0,
                            k_max_without_improvements=1000,
                            score_improvement_to_save=1.0,
                            score_min_to_save=-np.inf,
                            log_score_every=100,
                            return_on_restart=False)
    print("Best score after local search: {}".format(best_score))
    write_gph(best_G, vars, data_name=data_name, gph_name="best_after_optims",
score=best_score)
```