Process Discovery

Assignment 3

Name: KAMIL ILYAS
Roll No: 20I-2371
Section: P

Report

Part A

Step 1:

```
#reading the file using pd.read_csv
event_log = pd.read_csv('E:/Procmin/AnonymizedEventData.csv')

#grouping events in the event_log dataFrame based on 'TicketNum' column and then
#for each group extracting the 'Status' column and converting it into a list of events
traces = event_log.groupby('TicketNum')['Status'].apply(list).tolist()
L = [tuple(trace) for trace in traces]  #takes a list of traces and convert each trace into a tuple
print("Multi-set of traces (L):")
print(L)
```

This code takes an event log from a CSV file, organizes the events according to the trace, and then transforms the list of lists into a multi-set of traces that may be represented as a list of tuples. The Alpha algorithm uses this multi-set of traces to compute the set of event pairs that directly follow one another in the traces.

Output:

```
Multi-set of traces (L):
[('Open', 'acknowledged notification', 'Assignment', 'analysis/research/tech note', 'Assignment', 'pending customer', 'Assignment',
'reassigned-misrouted', 'reassigned-misrouted', 'acknowledged notification', 'Assignment', 'analysis/research/tech note', 'restored
to service', 'Closed'), ('Open', 'acknowledged notification', 'analysis/research/tech note', 'Closed'), ('Open', 'acknowledged
notification', 'analysis/research/tech note', 'Closed'), ('Open', 'Assignment', 'analysis/research/tech note', 'alert stage 1',
'alert stage 2', 'alert stage 3', 'DEADLINE ALERT', 'acknowledged notification', 'Assignment', 'analysis/research/tech note',
'restored to service', 'Closed', 'Closed', 'Closed'), ('Open', 'alert stage 1', 'alert stage 2', 'alert stage 3', 'DEADLINE ALERT',
'acknowledged notification', 'Assignment', 'analysis/research/tech note', 'restored to service', 'Closed', 'Closed', 'Closed'),
('Open', 'acknowledged notification', 'analysis/research/tech note', 'analysis/research/tech note', 'Closed', 'Closed'), ('Open',
'alert stage 1', 'alert stage 2', 'alert stage 3', 'Assignment', 'acknowledged notification', 'analysis/research/tech note',
'acknowledged notification', 'analysis/research/tech note', 'pending customer', 'Closed'), ('Open', 'alert stage 1', 'Assignment',
'Manually Acknowledged', 'analysis/research/tech note', 'restored to service', 'Closed'), ('Open', 'Assignment', 'acknowledged
notification', 'acknowledged notification', 'restored to service', 'Assignment', 'reassigned-addl work required', 'reassigned-addl
work required', 'reassigned-addl work required', 'alert stage 1', 'acknowledged notification', 'analysis/research/tech note',
'Closed', 'Closed'), ('Open', 'acknowledged notification', 'analysis/research/tech note', 'Assignment', 'analysis/research/tech
note', 'restored to service', 'Closed', 'Closed'), ('Open', 'acknowledged notification', 'Closed'), ('Open', 'acknowledged
notification', 'Assignment', 'analysis/research/tech note', 'reassigned-addl work required', 'reassigned-addl work required',
'acknowledged notification', 'Assignment', 'analysis/research/tech note', 'Closed', 'Closed'), ('Open', 'alert stage 1', 'alert
stage 2', 'alert stage 3', 'Assignment', 'acknowledged notification', 'analysis/research/tech note', 'analysis/research/tech note',
```

Step 2:

```
#contains all the unique statuses extracted from the list of traces L
TL = set(status for trace in L for status in trace)
print("Set TL (Unique Statuses):")
print(TL)
```

Using this code, a multi-set of traces' unique statuses are computed and displayed as a set. The process model, which is a directed graph that depicts the potential orderings of events in the log, is built using this set by the Alpha method. Events are represented by the graph's nodes, while event sequences are represented by the graph's edges.

Output:

```
Set TL:
{'pending vendor', 'Assignment', 'automated', 'alert stage 2', 'Pending Vendor', 'Acknowledged', 'reassigned-misrouted', 'pending
repair', 'Restored to Service', 'communication with provider', 'Pending Customer', 'Open', 'pending provider', 'alert stage 3',
'communication with customer', 'restarted notification', 'Closed', 'status request', 'communication with vendor', 'Work in
Progress', 'restored to service', 'pending customer', 'Manually Acknowledged', 'pending confirmation', 'alert stage 1', 'equipment
return', 'acknowledged notification', 'DEADLINE ALERT', 'reassigned-addl work required', 'pending release', 'analysis/research/tech
note', 'waiting parts'}
```

Step 3:

```
#iterating each trace in L and retrieving the first element using trace[0]
start_events = [trace[0] for trace in L]
end_events = [trace[-1] for trace in L]

#set is used to remove duplicates, max function is used to find elements that occur most frequently
TI = max(set(start_events), key=start_events.count)
TO = max(set(end_events), key=end_events.count)

print("Start event (TI):", TI)
print("End event (TO):", TO)
```

Based on the most common events that appear at the start and end of the traces in the multi-set of traces, this method computes the start event and end event of the process model. The process model, which is a directed graph that depicts the potential sequences of events in the log, is built using these events by the Alpha algorithm.

Output:

```
Start event (TI): Open
End event (TO): Closed
```

Step 4:

```
PL = set()  #initialized empty set
for trace in L:
    for i in range(1, len(trace)):   #iterates over all pairs of consecutive events
        predecessor = (trace[i-1], trace[i])   #creates a tuple of the form (a, b)
        PL.add(predecessor)   #adding predecessor tuple to the set

print("Set PL:")
for predecessor in PL:
    print(predecessor)
```

The set of all event pairs that directly follow one another in the traces is computed by this code, and the result is displayed as a set of tuples. The process model, which is a directed graph that depicts the potential orderings of events in the log, is built using this set by the Alpha method.

Output:

```
Set PL:
('communication with provider', 'communication with provider')
('automated', 'automated')
('acknowledged notification', 'restarted notification')
('waiting parts', 'status request')
('DEADLINE ALERT', 'reassigned-addl work required')
('pending release', 'Closed')
('status request', 'reassigned-addl work required')
('acknowledged notification', 'communication with customer')
('status request', 'DEADLINE ALERT')
('alert stage 2', 'DEADLINE ALERT')
('restored to service', 'DEADLINE ALERT')
('pending customer', 'reassigned-addl work required')
('pending vendor', 'pending confirmation')
('communication with provider', 'pending repair')
```

Step 5:

```
FL = set()
for trace in L:
    for i in range(len(trace) - 1):
        follow = (trace[i], trace[i+1])
        FL.add(follow)

print("Set FL:")
for follow in FL:
    print(follow)
```

This code computes the set of all event pairs that occur in the traces directly before one another, and then outputs the results as a set of tuples. The process model, which is a directed graph that depicts the potential orderings of events in the log, is built using this set by the Alpha method.

Output:

```
Set FL:
('Manually Acknowledged', 'pending vendor')
('pending release', 'analysis/research/tech note')
('acknowledged notification', 'equipment return')
('pending repair', 'pending customer')
('pending repair', 'Assignment')
('status request', 'Closed')
('Manually Acknowledged', 'pending confirmation')
('communication with customer', 'reassigned-addl work required')
('alert stage 2', 'pending vendor')
('waiting parts', 'pending customer')
('alert stage 1', 'alert stage 2')
('waiting parts', 'Assignment')
('analysis/research/tech note', 'restarted notification')
('reassigned-misrouted', 'Closed')
('communication with customer', 'restarted notification')
('reassigned-addl work required', 'Assignment')
('acknowledged notification', 'pending customer')
('acknowledged notification', 'waiting parts')
```

Step 6:

```
resultant_process = Digraph()
resultant_process.attr(rankdir='LR')
resultant_process.attr('node', shape='rectangle')
resultant_process.node(TI)
for relation in PL:
    resultant_process.edge(relation[0], relation[1])
resultant_process.node(TO)

print("Resultant Process:")
print(resultant_process.source)
```

Using the graphviz package, the following code generates a directed graph. It establishes the nodes' rectangular shapes and the graph's left-to-right orientation. Then, based on a set of relations, it adds nodes to the network and edges between them. The output graph is then printed to the console as a string.

Output:

```
Resultant Process:
digraph {
  rankdir=LR
  node [shape=rectangle]
  Open
  "communication with customer" -> "pending release"
  "acknowledged notification" -> "waiting parts"
  "analysis/research/tech note" -> "DEADLINE ALERT"
  "pending vendor" -> "analysis/research/tech note"
  "status request" -> "pending repair"
  "pending confirmation" -> "restored to service"
  "communication with provider" -> "restored to service"
  "alert stage 3" -> "DEADLINE ALERT"
  "restored to service" -> "pending release"
  "reassigned-addl work required" -> "pending release"
  "alert stage 2" -> "alert stage 3"
  Closed -> "status request"
  "analysis/research/tech note" -> "Work in Progress"
  "pending repair" -> "acknowledged notification"
  "pending customer" -> "pending repair"
  "reassigned-misrouted" -> "communication with customer"
```

Visualizing process:

```
resultant_process = nx.DiGraph()
resultant_process.add_edges_from(FL)

plt.figure(figsize=(8, 6))
pos = nx.spring_layout(resultant_process)
nx.draw(resultant_process, pos, with_labels=True, node_color='lightblue', edge_color='gray', arrows=True)
plt.title("Resultant Process")
plt.savefig("resultant_process.png")
plt.show()
```

This code uses networkx and matplotlib to visualise a directed graph. The plot is created with labels, light blue nodes, grey margins, and arrows. The plot's title is changed to "Resultant Process" before being displayed and saved as a PNG file.

Output:

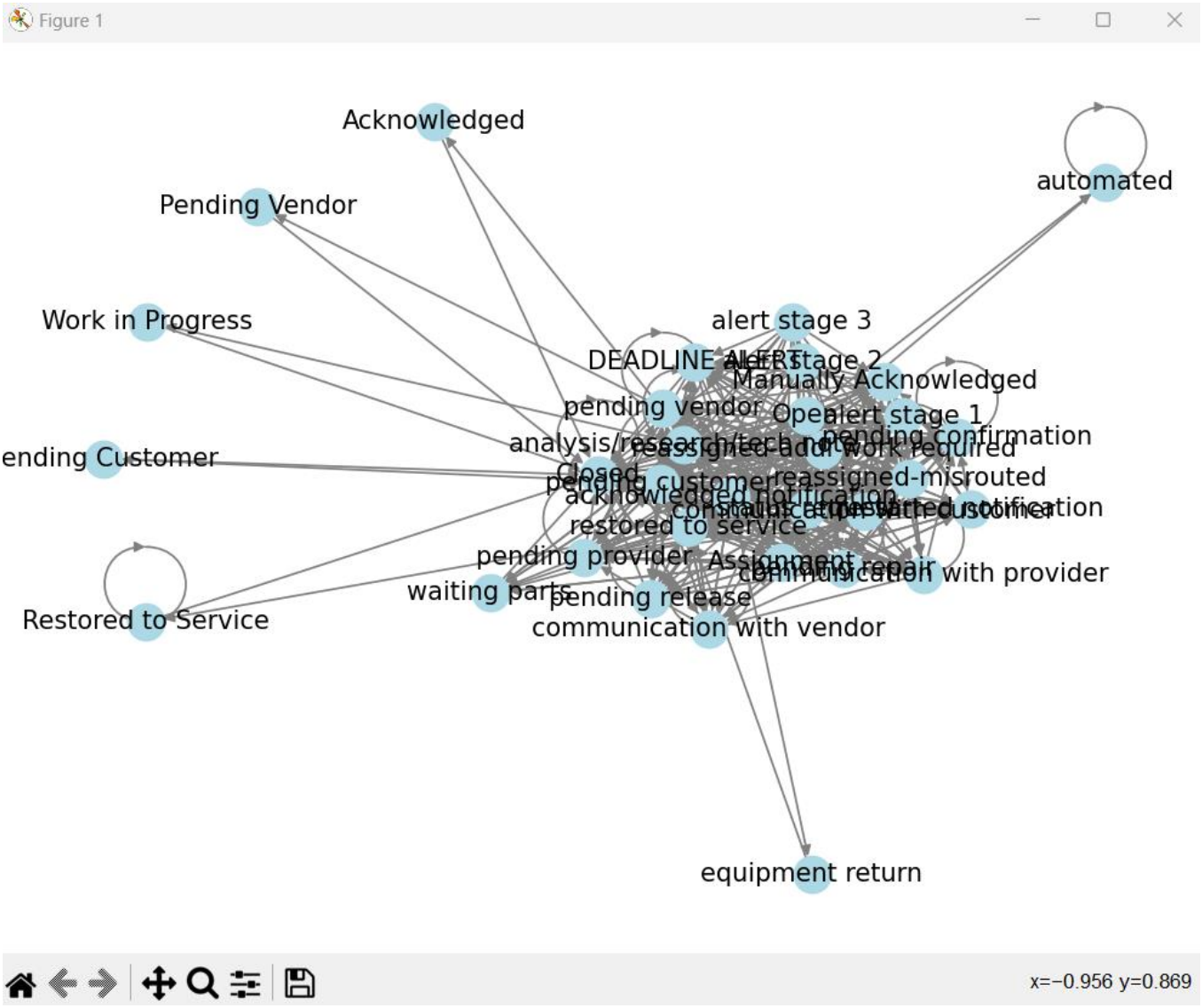
Figure 1
Acknowledged
Pending Vendor
Work in Progress
ending Customer
Restored to Service
automated
alert stage 3
DEADLINE ALERT
Manually Acknowledged
pending vendor
Open
pending confirmation
reassigned-misrouted
pending customer
restored to service
pending provider
waiting parts
pending release
communication with vendor
equipment return
x=−0.956 y=0.869

Part B

```
fitness_results = []

for trace in L:
    current_node = trace[0]
    executed_trace = [current_node]

    for event in trace[1:]:
        if (current_node, event) in FL:
            current_node = event
            executed_trace.append(current_node)

    if executed_trace == list(trace):
        fitness_results.append((trace, True))  #trace successfully executed on the process
    else:
        fitness_results.append((trace, False))  #trace did not be executed on the process


print("Fitness Results:")
for trace, is_executable in fitness_results:
    print(f"Trace: {trace}, Executable: {is_executable}")
```

By contrasting it with a collection of event logs, this code determines the fitness of a process model. It starts a list called fitness_results that is empty. The function iterates over the events in the trace for each trace in the event logs (L), determining if the next event can be processed based on the previous event and the process model (FL). The event is added to the executed_trace list if it can be carried out. The trace is deemed executable if the executed_trace list corresponds to the original trace, in which case a tuple with the trace and the boolean value True is added to the fitness_results list. In the absence of it, the fitness_results list is supplemented with a tuple that contains the trace and the boolean value False.

Output:

```
Fitness Results:
Trace: ('Open', 'acknowledged notification', 'Assignment', 'analysis/research/tech note', 'Assignment', 'pending customer',
'Assignment', 'reassigned-misrouted', 'reassigned-misrouted', 'acknowledged notification', 'Assignment', 'analysis/research/tech
note', 'restored to service', 'Closed'), Executable: True
Trace: ('Open', 'acknowledged notification', 'analysis/research/tech note', 'Closed'), Executable: True
Trace: ('Open', 'acknowledged notification', 'analysis/research/tech note', 'Closed'), Executable: True
Trace: ('Open', 'Assignment', 'analysis/research/tech note', 'alert stage 1', 'alert stage 2', 'alert stage 3', 'DEADLINE ALERT',
'acknowledged notification', 'Assignment', 'analysis/research/tech note', 'restored to service', 'Closed', 'Closed', 'Closed'),
Executable: True
Trace: ('Open', 'alert stage 1', 'alert stage 2', 'alert stage 3', 'DEADLINE ALERT', 'acknowledged notification', 'Assignment',
'analysis/research/tech note', 'restored to service', 'Closed', 'Closed', 'Closed'), Executable: True
Trace: ('Open', 'acknowledged notification', 'analysis/research/tech note', 'analysis/research/tech note', 'Closed', 'Closed'),
Executable: True
Trace: ('Open', 'alert stage 1', 'alert stage 2', 'alert stage 3', 'Assignment', 'acknowledged notification', 'analysis/research/
tech note', 'acknowledged notification', 'analysis/research/tech note', 'pending customer', 'Closed'), Executable: True
Trace: ('Open', 'alert stage 1', 'Assignment', 'Manually Acknowledged', 'analysis/research/tech note', 'restored to service',
'Closed'), Executable: True
Trace: ('Open', 'Assignment', 'acknowledged notification', 'acknowledged notification', 'restored to service', 'Assignment',
'reassigned-addl work required', 'reassigned-addl work required', 'reassigned-addl work required', 'alert stage 1', 'acknowledged
notification', 'analysis/research/tech note', 'Closed', 'Closed'), Executable: True
```