CS5300 Assignment5

Name: Ke Wang
NetId: kw427

1. SAGAs and Trust

Part (a): In that case, the sequence would be $T_1T_2C_1$. This is because if T_2 fails; we have to roll back the transactions. T_2 is the last step and it has no compensation action since server state wouldn't be modified if it fails. So we only need to compensate T_1.

Part (b): This scenario would happen when Bob withdrew all the money in this account from the bank before he delivers the widget. In that case, he can "cheat" that when T_2 fails, there is no money in his account; i.e. he keeps Alice's money without delivering the widget.
This can also be put as: Add a new action T_3: Bob withdrew all the money from his account. Then we have two sequences of transactions. A is T_1T_2, and the other B is T_3. They should be executed as the following sequence where T_3 is executed between T_1 and T_2. Since T_3 is in different transaction from B, which doesn't need to be rolled back when A is rolled back.

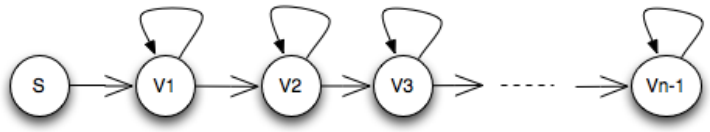
(A)	(B)
T_1	
	T_3
T_2	

Part (c): We can modify the SAGA by adding Carol as a coordinator. The new actions would be:
T_1: balance[Alice] -= \$100; balance[Carol] += \$100
C_1: balance[Carol] -= \$100; balance[Alice] += \$100
T_2: Bob delivers the widget to Alice; widgets[Bob] -= 1; widgets[Alice] += 1
C_2: Alice delivers back the widgets to Bob; widgets[Alice] -= 1; widgets[Bob] += 1
T_3: balance[Carol] -= \$100; balance[Bob] += \$100

Then we execute $T_1T_2T_3$. Bob cannot cheat in this case. Bob can get his money (T_3) from Carol only after he successfully delivered the widgets to Alice (T_2). Carol is famous for her honesty and she can keep the money and widgets so the transaction can be ensured.

2.Blocked Shortest Path in MapReduce

Part (a): The example is in the picture. The method stated in the question actually describes Dijkstra's Algorithm and it is breadth-first. In our case these nodes are sequentially ordered where we can only modify the distance of at most one node for each MapReduce pass (nodes after that are still with $d(v) = \infty$). Totally, it requires $\Omega(N)$ passes to converge.



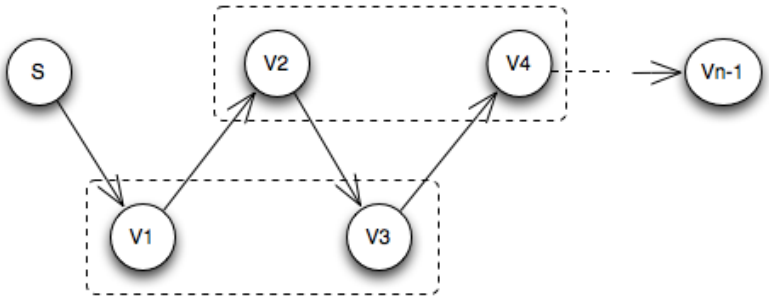
Part (b): Reducer for block B:

Receiving every edge that enters or leaves a node in B. There are three kinds of nodes in block B: (1) node that is the endpoint of edge enters in B; (2) node that is the endpoint of edge leaves B; (3) internal nodes without edges in or out of B.

Reducer first compute the new distance $d'(v)$ for the nodes of the first kind with $d'(v) = \min(\{d(v)\} \cup \{d(u) + 1 \mid u \to v\})$. Then compute the new distances $d'(v)$ for all the other nodes in block B until the values converged (e.g. by Dijkstra's Algorithm). Then the reducer emits:
<$blockNum(u)$; $u \to v$, $d(u)$> and <$blockNum(v)$; $u \to v, d(u)$> for all edges in the block where the $d(u)$ are the new values updated after this pass.
In this case, we can minimize the number of MapReduce passes.

Part (c): If the master can remember the assignments for each Reducer and each Reducer can put nodes and edges information in the memory, we can omit edges within a Block. That is, if there is an edge $u \to v$ within a block, the Mapper can omit message of <$blockNum(u)$; $u \to v, d(u)$> and <$blockNum(v)$; $u \to v, d(u)$>. This is because these messages would not affect the input/output of the endpoints of edges between blocks. The information of edges within blocks are recorded in the Reducer memory, so that it can be used to compute the new distances within the block in each pass.

Part (d): We can use the graph from part (a) and omit the self-loop without loss of generality. The idea is to choose poorly partitions of blocks, e.g. nodes in the same block are disconnected. For example, block size is 2 and the rectangle is a block, the nodes can be divided as follows: Though with improved Blocked implementation, it still requires $\Omega(N)$ MapReduce passes to converge.


S
V2
V4
Vn-1
V1
V3