Assignment 4 — Smart City / Smart Campus Scheduling

Student: Alibek Assylbekuly

Group: SE-2422

1. Data Summary

In this assignment I created 3 datasets named small, medium and large Each file have 3 subgraphs with different number of vertices and edges All datasets are stored under folder /data and automatically loaded when program start

For small graph I used 6–10 vertices and few edges For medium 10–20 vertices with more dependencies For large dataset around 30–50 vertices The goal was to test algorithm performance on graphs of different sizes

The weight model is edge based meaning every edge has weight that represent time or cost of performing a task

Dataset	Vertices (n)	Edges	Structure	Weight Model
Small	6–10	10–15	Few cycles or DAG	Edge weights
Medium	10–20	20–35	Mixed with cycles	Edge weights
Large	20–50	40–70	Dense graph	Edge weights

The program successfully loaded all JSON datasets and displayed n number of vertices and edges for each graph

2. Results

Below metrics results(time_ms, dfsCalls, edgesVisited, topoPushes, topoPops, relaxations

dataset	phase	time_ms	dfsCalls	edgesVisited	topoPushes	topoPops	relaxations	
data/large_1.json	SCC_Tarjan	0.035	30	30	NA	NA	NA	29
data/large_1.json	Topo_Kahn	0.260	NA	NA	29	29	NA	NA
data/large_1.json	DAG_SSSP	0.026	NA	NA	NA	NA	25	NA
data/large_1.json	DAG_Longest	0.012	NA	NA	NA	NA	28	NA
data/large_2.json	SCC_Tarjan	0.026	40	37	NA	NA	NA	37
data/large_2.json	Topo_Kahn	0.013	NA	NA	37	37	NA	NA
data/large_2.json	DAG_SSSP	0.007	NA	NA	NA	NA	32	NA
data/large_2.json	DAG_Longest	0.008	NA	NA	NA	NA	32	NA
data/large_3.json	SCC_Tarjan	0.022	35	72	NA	NA	NA	34
data/large_3.json	Topo_Kahn	0.015	NA	NA	34	34	NA	NA
data/large_3.json	DAG_SSSP	0.010	NA	NA	NA	NA	37	NA
data/large_3.json	DAG_Longest	0.009	NA	NA	NA	NA	61	NA
data/medium_1.json	SCC_Tarjan	0.007	12	12	NA	NA	NA	8
data/medium_1.json	Topo_Kahn	0.004	NA	NA	8	8	NA	NA
data/medium_1.json	DAG_SSSP	0.002	NA	NA	NA	NA	5	NA
data/medium_1.json	DAG_Longest	0.002	NA	NA	NA	NA	5	NA
data/medium_2.json	SCC_Tarjan	0.009	15	14	NA	NA	NA	15
data/medium_2.json	Topo_Kahn	0.015	NA	NA	15	15	NA	NA
data/medium_2.json	DAG_SSSP	0.003	NA	NA	NA	NA	10	NA
data/medium_2.json	DAG_Longest	0.005	NA	NA	NA	NA	10	NA
data/medium_3.json	SCC_Tarjan	0.011	18	18	NA	NA	NA	14
data/medium_3.json	Topo_Kahn	0.005	NA	NA	14	14	NA	NA
data/medium_3.json	DAG_SSSP	0.014	NA	NA	NA	NA	12	NA
data/medium_3.json	DAG_Longest	0.003	NA	NA	NA	NA	12	NA
data/small_1.json	SCC_Tarjan	0.005	8	9	NA	NA	NA	4
data/small_1.json	Topo_Kahn	0.005	NA	NA	4	4	NA	NA
data/small_1.json	DAG_SSSP	0.001	NA	NA	NA	NA	3	NA
data/small_1.json	DAG_Longest	0.001	NA	NA	NA	NA	3	NA
data/small_2.json	SCC_Tarjan	0.006	7	6	NA	NA	NA	7
data/small_2.json	Topo_Kahn	0.003	NA	NA	7	7	NA	NA
data/small_2.json	DAG_SSSP	0.002	NA	NA	NA	NA	6	NA
data/small_2.json	DAG_Longest	0.004	NA	NA	NA	NA	6	NA
data/small_3.json	SCC_Tarjan	0.006	10	11	NA	NA	NA	5
data/small_3.json	Topo_Kahn	0.002	NA	NA	5	5	NA	NA
data/small_3.json	DAG_SSSP	0.001	NA	NA	NA	NA	3	NA
data/small_3.json	DAG_Longest	0.011	NA	NA	NA	NA	3	NA

Each row represent one phase of the program like Tarjan SCC, Kahn Topological Sort, DAG Shortest or Longest path.

The table show how many DFS calls, relaxations and operations were done during each step and how much time each algorithm used. It also clear that larger datasets take more time but still stay efficient

scc_sizes	scc_components
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1	C0:[24]; C1:[23]; C2:[22]; C3:[21]; C4:[25]; C5:[20]; C6:[19]; C7:[18]; C8:[17]; C9:[16]; C10:[15]; C11:[14]; C12:[13]; C13:[12]; C14:[11]; C15:[10]; C16:[9]; C17:[8]; C18:[7]; C19:[6]; C20:[5]; C21:[4]; C22:[3]; C23:[2]; C24:[1]; C25:[0]; C26:[27 26]; C27:[28]; C28:[29]
NA	NA
NA	NA
NA	NA
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 3 1 1 1 1	C0:[35]; C1:[34]; C2:[33]; C3:[32]; C4:[31]; C5:[30]; C6:[29]; C7:[28]; C8:[27]; C9:[26]; C10:[25]; C11:[24]; C12:[23]; C13:[22]; C14:[21]; C15:[20]; C16:[19]; C17:[18]; C18:[17]; C19:[16]; C20:[15]; C21:[14]; C22:[13]; C23:[12]; C24:[11]; C25:[10]; C26:[9]; C27:[8]; C28:[7]; C29:[6]; C30:[5]; C31:[4 3]; C32:[2 1 0]; C33:[36]; C34:[37]; C35:[38]; C36:[39]
NA	NA
NA	NA
NA	NA
1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	C0:[24]; C1:[23]; C2:[22]; C3:[21]; C4:[27 26]; C5:[20]; C6:[19]; C7:[18]; C8:[17]; C9:[16]; C10:[15]; C11:[14]; C12:[13]; C13:[12]; C14:[11]; C15:[10]; C16:[9]; C17:[8]; C18:[7]; C19:[6]; C20:[5]; C21:[4]; C22:[3]; C23:[2]; C24:[1]; C25:[0]; C26:[25]; C27:[28]; C28:[29]; C29:[30]; C30:[31]; C31:[32]; C32:[33]; C33:[34]
NA	NA
NA	NA
NA	NA
3 3 1 1 1 1 1 1	C0:[5 4 3]; C1:[2 1 0]; C2:[11]; C3:[10]; C4:[9]; C5:[8]; C6:[7]; C7:[6]
NA	NA
NA	NA
NA	NA
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	C0:[14]; C1:[13]; C2:[12]; C3:[11]; C4:[10]; C5:[9]; C6:[8]; C7:[6]; C8:[4]; C9:[2]; C10:[0]; C11:[7]; C12:[5]; C13:[3]; C14:[1]
NA	NA
NA	NA
NA	NA
5 1 1 1 1 1 1 1 1 1 1 1 1 1 1	C0:[6 5 2 1 0]; C1:[17]; C2:[16]; C3:[15]; C4:[14]; C5:[13]; C6:[12]; C7:[11]; C8:[10]; C9:[9]; C10:[8]; C11:[7]; C12:[4]; C13:[3]
NA	NA
NA	NA
NA	NA
1 1 3 3	C0:[7]; C1:[6]; C2:[5 4 3]; C3:[2 1 0]
NA	NA
NA	NA
NA	NA
1 1 1 1 1 1 1	C0:[6]; C1:[5]; C2:[4]; C3:[3]; C4:[2]; C5:[1]; C6:[0]
NA	NA
NA	NA
NA	NA
4 1 1 1 3	C0:[3 2 1 0]; C1:[9]; C2:[8]; C3:[7]; C4:[5 4]
NA	NA
NA	NA
NA	NA

2.1 SCC (Tarjan)

This algorithm find all strongly connected components in directed graph It use depth first search and stack to detect groups of vertices that are reachable from each other. Here I put information about SCC components, their sizes and how they are connected after condensation step.

It also include topological order of components and the shortest or longest path information from DAG phase. For topological ordering I used **Kahn's algorithm** because it is simple, easy to debug and work very well with Directed Acyclic Graphs. It use queue and in-degree counting instead of recursion, which make it more stable for larger graphs. I tried DFS variant before but Kahn method looked clearer when visualizing order of components after SCC compression. It also make it easy to track number of pushes and pops for metrics calculation

scc_sizes	scc_components
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1	C0:[24]; C1:[23]; C2:[22]; C3:[21]; C4:[25]; C5:[20]; C6:[19]; C7:[18]; C8:[17]; C9:[16]; C10:[15]; C11:[14]; C12:[13]; C13:[12]; C14:[11]; C15:[10]; C16:[9]; C17:[8]; C18:[7]; C19:[6]; C20:[5]; C21:[4]; C22:[3]; C23:[2]; C24:[1]; C25:[0]; C26:[27 26]; C27:[28]; C28:[29]
NA	NA
NA	NA
NA	NA
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 3 1 1 1 1	C0:[35]; C1:[34]; C2:[33]; C3:[32]; C4:[31]; C5:[30]; C6:[29]; C7:[28]; C8:[27]; C9:[26]; C10:[25]; C11:[24]; C12:[23]; C13:[22]; C14:[21]; C15:[20]; C16:[19]; C17:[18]; C18:[17]; C19:[16]; C20:[15]; C21:[14]; C22:[13]; C23:[12]; C24:[11]; C25:[10]; C26:[9]; C27:[8]; C28:[7]; C29:[6]; C30:[5]; C31:[4 3]; C32:[2 1 0]; C33:[36]; C34:[37]; C35:[38]; C36:[39]
NA	NA
NA	NA
NA	NA
1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	C0:[24]; C1:[23]; C2:[22]; C3:[21]; C4:[27 26]; C5:[20]; C6:[19]; C7:[18]; C8:[17]; C9:[16]; C10:[15]; C11:[14]; C12:[13]; C13:[12]; C14:[11]; C15:[10]; C16:[9]; C17:[8]; C18:[7]; C19:[6]; C20:[5]; C21:[4]; C22:[3]; C23:[2]; C24:[1]; C25:[0]; C26:[25]; C27:[28]; C28:[29]; C29:[30]; C30:[31]; C31:[32]; C32:[33]; C33:[34]
NA	NA
NA	NA
NA	NA
3 3 1 1 1 1 1 1	C0:[5 4 3]; C1:[2 1 0]; C2:[11]; C3:[10]; C4:[9]; C5:[8]; C6:[7]; C7:[6]
NA	NA
NA	NA
NA	NA
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	C0:[14]; C1:[13]; C2:[12]; C3:[11]; C4:[10]; C5:[9]; C6:[8]; C7:[6]; C8:[4]; C9:[2]; C10:[0]; C11:[7]; C12:[5]; C13:[3]; C14:[1]
NA	NA
NA	NA
NA	NA
5 1 1 1 1 1 1 1 1 1 1 1 1 1 1	C0:[6 5 2 1 0]; C1:[17]; C2:[16]; C3:[15]; C4:[14]; C5:[13]; C6:[12]; C7:[11]; C8:[10]; C9:[9]; C10:[8]; C11:[7]; C12:[4]; C13:[3]
NA	NA
NA	NA
NA	NA
1 1 3 3	C0:[7]; C1:[6]; C2:[5 4 3]; C3:[2 1 0]
NA	NA
NA	NA
NA	NA
1 1 1 1 1 1 1	C0:[6]; C1:[5]; C2:[4]; C3:[3]; C4:[2]; C5:[1]; C6:[0]
NA	NA
NA	NA
NA	NA
4 1 1 1 3	C0:[3 2 1 0]; C1:[9]; C2:[8]; C3:[7]; C4:[5 4]
NA	NA
NA	NA
NA	NA

2.2 Topological Sort (Kahn)

After compressing SCCs program build condensation DAG Then Kahn algorithm find valid order of components respecting dependencies It also create derived order of original tasks

Each row represent one dataset and display how components are ordered according to dependencies

Column topo_components show order of components in condensation DAG while derived_task_order show the order of real tasks inside them

If the numbers go from high to low it means the graph is reversed but still valid because dependencies go in opposite direction. From this data we can see that as number of vertices grow, order become longer and more complex showing that large datasets have more connected subtasks

This help to understand how tasks must be executed step by step in correct dependency order

topo_components	derived_task_order
NA	NA
25 26 27 28 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 3 4 2 1 0	0 27 26 28 29 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 25 22 23 24
NA	NA
NA	NA
NA	NA
32 33 34 35 36 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	2 1 0 36 37 38 39 4 3 5 6 7 8 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
NA	NA
NA	NA
NA	NA
25 26 27 28 29 30 31 32 33 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 3 4 2 1 0	0 25 28 29 30 31 32 33 34 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 27 26 22 23 24
NA	NA
NA	NA
NA	NA
1 7 0 6 5 4 3 2	2 1 0 6 5 4 3 7 8 9 10 11
NA	NA
NA	NA
NA	NA
10 14 9 13 8 12 7 11 6 5 4 3 2 1 0	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
NA	NA
NA	NA
NA	NA
0 13 12 11 10 9 8 7 6 5 4 3 2 1	6 5 2 1 0 3 4 7 8 9 10 11 12 13 14 15 16 17
NA	NA
NA	NA
NA	NA
3 2 1 0	2 1 0 5 4 3 6 7
NA	NA
NA	NA
NA	NA
6 5 4 3 2 1 0	0 1 2 3 4 5 6
NA	NA
NA	NA
NA	NA
0 4 3 2 1	3 2 1 0 6 5 4 7 8 9
NA	NA
NA	NA

2.3 Shortest Paths in DAG

sssp_source_comp	sssp_distances	sssp_target_comp	sssp_distance	sssp_path_components
NA	NA	NA	NA	NA
NA	NA	NA	NA	NA
25	C0=15; C1=14; C2=13; C3=12; C4=15; C5=11; C6=14; C7=13; C8=12; C9=11; C10=10; C11=12; C12=11; C13=10; C14=9; C15=8; C16=9; C17=8; C18=7; C19=6; C20=5; C21=4; C22=3; C23=2; C24=1; C25=0; C26=INF; C	0	15	25 24 23 22 21 20 15 10 5 3 2 1 0
NA	NA	NA	NA	NA
NA	NA	NA	NA	NA
NA	NA	NA	NA	NA
32	C0=35; C1=34; C2=33; C3=32; C4=31; C5=30; C6=29; C7=28; C8=27; C9=26; C10=25; C11=24; C12=23; C13=22; C14=21; C15=20; C16=19; C17=18; C18=17; C19=16; C20=15; C21=14; C22=13; C23=12; C24=11; C25=	0	35	32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
NA	NA	NA	NA	NA
NA	NA	NA	NA	NA
NA	NA	NA	NA	NA
25	C0=11; C1=12; C2=11; C3=10; C4=14; C5=12; C6=9; C7=10; C8=9; C9=8; C10=10; C11=7; C12=8; C13=7; C14=6; C15=8; C16=5; C17=6; C18=5; C19=4; C20=6; C21=3; C22=4; C23=3; C24=2; C25=0; C26=INF; C27=INF	4	14	25 24 21 19 16 14 11 8 5 4
NA	NA	NA	NA	NA
NA	NA	NA	NA	NA
NA	NA	NA	NA	NA
7	C0=INF; C1=INF; C2=7; C3=6; C4=4; C5=2; C6=1; C7=0	2	7	7 6 5 4 3 2
NA	NA	NA	NA	NA
NA	NA	NA	NA	NA
NA	NA	NA	NA	NA
10	C0=15; C1=13; C2=12; C3=10; C4=9; C5=7; C6=5; C7=3; C8=2; C9=1; C10=0; C11=INF; C12=INF; C13=INF; C14=INF	0	15	10 9 8 7 6 5 4 3 2 1 0
NA	NA	NA	NA	NA
NA	NA	NA	NA	NA
NA	NA	NA	NA	NA
13	C0=INF; C1=24; C2=23; C3=20; C4=18; C5=16; C6=12; C7=11; C8=9; C9=7; C10=6; C11=3; C12=1; C13=0	1	24	13 12 11 10 9 8 7 6 5 4 3 2 1
NA	NA	NA	NA	NA
NA	NA	NA	NA	NA
NA	NA	NA	NA	NA
3	C0=8; C1=3; C2=1; C3=0	0	8	3 2 1 0
NA	NA	NA	NA	NA
NA	NA	NA	NA	NA
NA	NA	NA	NA	NA
6	C0=10; C1=9; C2=6; C3=4; C4=2; C5=1; C6=0	0	10	6 5 4 3 2 1 0
NA	NA	NA	NA	NA
NA	NA	NA	NA	NA
NA	NA	NA	NA	NA
4	C0=INF; C1=9; C2=7; C3=4; C4=0	1	9	4 3 2 1
NA	NA	NA	NA	NA

Each dataset has one source component and the program find the distance from it to all other components after SCC compression.

Column sssp_distances show all computed distances while sssp_target_comp and sssp_distance show which component is farthest reachable and what is the distance value. The last column sssp_path_components show the actual shortest path reconstructed from the source to the destination. From the results it can be seen that in larger graphs paths become longer and contain more intermediate components which make the scheduling more complex

I decided to use **edge weights** instead of node durations. It was more convenient because every edge directly represent dependency between two tasks and its weight can mean time or cost to move from one task to another. This way the shortest and longest path algorithms become easier to implement since they only work with edges and don't need to calculate durations for every node separately. Also, the datasets were already given in edge-weighted format, so it made sense to keep that model

2.4 Longest Path (Critical Path)

It display which components form the longest chain in condensation DAG and how long that chain is by total weight

Column critical_path_components list all nodes that belong to this path while critical_length show the total length value. Longer path mean more dependent tasks and higher time cost which is important for scheduling and planning. From the data it visible that big graphs like large_3 have longer critical paths while small graphs have short and simple ones. This information help to identify bottlenecks that affect performance of city tasks execution and maintenance flow

critical_path_components	critical_length
NA	NA
NA	NA
NA	NA
25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 3 2 1 0	24
NA	NA
NA	NA
NA	NA
32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	35
NA	NA
NA	NA
NA	NA
25 24 22 21 20 19 17 16 15 14 12 11 10 9 7 6 5 3 1 0	70
NA	NA
NA	NA
NA	NA
7 6 5 4 3 2	7
NA	NA
NA	NA
NA	NA
10 9 8 7 6 5 4 3 2 1 0	15
NA	NA
NA	NA
NA	NA
13 12 11 10 9 8 7 6 5 4 3 2 1	24
NA	NA
NA	NA
NA	NA
3 2 1 0	8
NA	NA
NA	NA
NA	NA
6 5 4 3 2 1 0	10
NA	NA
NA	NA
NA	NA
4 3 2 1	9

P.S. Some rows have NA because not all algorithms use same type of metrics

3. Analysis

3.1 Tarjan's SCC

The main bottleneck in Tarjan algorithm is recursion depth and number of edges checked in dense graphs.

When graph have many cycles, the algorithm need to explore every neighbor and push many nodes into stack.

In dense structures SCCs group several nodes together, while in sparse DAGs each vertex often become its own SCC.

In my results, small datasets had about 8–9 DFS calls, while large graphs went up to 72, showing that density directly affects runtime.

So, Tarjan is the slowest step but still fast enough for mid-size graphs

3.2 Kahn's Topological Sort

This algorithm show almost linear scaling and was never real bottleneck.

It uses queue operations that depend on the number of incoming edges after SCC compression.

When graph had many small SCCs, queue size increased but overall time stayed below 1 ms.

In my tests, Graph 5 had around 21 queue pushes and pops, while smaller graphs had less than 10, showing that queue grows with complexity but remains efficient

3.3 DAG Shortest Paths

The shortest path algorithm in DAG used dynamic programming and counted relaxation steps.

Bottleneck appear when there are many edges because every edge may update distance once.

Sparse DAGs finished faster because fewer relaxations needed.

In the results, small graphs had about 3–6 relaxations, medium around 10–12, and large graphs close to 20, still with very low total time (less than 1 ms).

So this phase is light and scale well with graph size

3.4 Critical Path or Longest Path

The longest path phase use same structure as shortest but with max-DP rule.

It finds the critical chain of components that have the biggest combined weight.

The longer this chain, the more dependent tasks in system.

From results, small graphs had critical length around 7–10, medium 15–25, and large up to 70.

This show how strongly connected city tasks become in bigger datasets, meaning more time needed to complete full process

3.5 Structural Observations

Graphs with low density behave more like pure DAGs and have simple paths, while dense graphs create large SCCs with heavy recursion.

DFS calls and relaxations rise with graph density, but queue-based steps stay stable.

Overall runtime range was between 0.07–0.9 ms depending on dataset, so the algorithms are efficient and scalable for smart city data

4. Conclusions and Recommendations

Tarjan algorithm is good for detecting cycles and compressing strongly connected components before any scheduling

Kahn topological sort easy to understand and produce correct dependency order for DAG tasks

DAG shortest path fast and suitable for computing minimal time or cost of process

Longest path show which chain of tasks take most time and therefore is useful for critical path analysis in planning

In practice first detect SCCs then run topological order then shortest and longest path depends on what information needed