Behavioral Equilibrium Solver (Network-Wide) - v5.4

How to run all 11 behavioral-equilibrium variants on Sioux Falls (CSV inputs), and how to extract every metric from the full trace bundles (BE vs UE vs SO, group splits, beliefs, utilities, flows).

Generated: 2026-02-04

Main solver module	**helper_scripts_v5_4_networkwide_trace.py**
Example runner (CSV)	run_sioux_falls_all_models_from_csv_v4.py (included in Appendix A)
Outputs per model	result.pkl.gz (full) + result_summary.json (light) + model_meta.json

Key deliverable for your paper: you can build a single summary table with columns TSTT_BE, TSTT_UE, TSTT_SO (and deltas), directly from the saved bundles. Section 7 gives copy/paste code.

1. What this code is doing - in project terms

Your project is about using information (signals) and behavioral decision models to influence route choice, then checking whether that changes system outcomes (especially total system travel time, TSTT) relative to classic baselines like UE and SO.

Instead of assuming travelers know the network perfectly (UE) or cooperate to minimize total delay (SO), you model two populations:

- **Uninformed group**: holds a *prior* belief about demand uncertainty.
- **Informed group**: receives a signal and updates its belief (or linearly pools beliefs), then chooses routes accordingly.

Both groups then play a behavioral equilibrium game: each group chooses path probabilities; those choices create link congestion; congestion changes perceived/experienced travel times; and the groups respond until the strategies stabilize.

The point of running all 11 variants is to quantify how outcomes change when you swap the behavioral model (Prospect/Variance vs ARA vs RRA) and when you change the reference point used to define gains/losses or deviations (UE vs previous-iteration/behavioral vs free-flow scaled).

Where UE and SO fit in

Every run computes three network states on the same K-path subnetwork:

- **UE** (User Equilibrium benchmark): travelers selfishly minimize their own travel time, solved here with a simple MSA loop on the restricted K-paths.
- **SO** (System Optimum benchmark): flows minimize total system travel time, approximated here via MSA on marginal costs on the same restricted K-paths.
- **BE** (Behavioral Equilibrium): two-group equilibrium under uncertainty + signals + behavioral utility model.

This controlled comparison (BE vs UE vs SO) is exactly what reviewers will look for when you claim information/behavior changes system performance.

2. Solver algorithms (technical, but readable)

2.1 Path sets and the restricted K-path subnetwork

The solver does not search over all possible paths in the full graph. Instead, it assumes each OD pair has a fixed set of K candidate paths (K-shortest paths). All assignment and equilibria are computed on this restricted set. If K is small, UE/SO/BE are approximations of the true full-network solutions.

2.2 Link performance function

Link travel time is computed using a BPR-style function on each directed link (u,v):

```
t_e(x_e) = t0_e * ( 1 + alpha * (x_e / cap_e)^beta )
```

In the Sioux Falls CSVs, t0 = free-flow time, cap = capacity, alpha/beta are the BPR parameters per link (or shared).

2.3 UE and SO benchmarks: MSA (not Frank-Wolfe)

This implementation does not use Frank-Wolfe. UE and SO are computed using Method of Successive Averages (MSA) on the fixed K-path sets.

- **UE-MSA step**: compute current link times, find each OD's shortest path (among its K), do an all-or-nothing assignment, then update path flows with step size 1/k.
- **SO-MSA step**: same outer loop, but shortest paths are computed on marginal costs: t + x * dt/dx.

2.4 Behavioral Equilibrium (BE) game loop

BE is solved as an iterative fixed-point game between uninformed (U) and informed (I) travelers:

```
Given: OD base demands D_od and a belief over theta (global demand multiplier).

At each BE iteration:
  1) For each group g in {U, I}:
       - Compute expected utility for each OD's K paths by integrating over theta-grid scenarios.
       - Convert utilities to a target path-choice probability vector q_g(od) using softmax/logit.
  2) Mix the two groups by market penetration m:
       q_mix = (1-m) * q_U + m * q_I
       path_flows_total(od,k) = D_real_od * q_mix(od,k)
  3) Compute link flows/times from path flows; this defines the realized network state.
  4) Update q_U and q_I toward their targets with an MSA step (1/iteration).
  5) Stop when q changes are below tolerance or max iterations reached.
```

So BE uses MSA to update strategies and uses a numerically integrated expected-utility calculation inside each iteration.

3. Beliefs over demand uncertainty (theta) and the PDF floor

Uncertainty is modeled as a single global multiplier theta applied to every OD demand: D_real_od = theta_real * D_base_od. Beliefs are distributions over theta (Uniform/Beta/Normal etc.).

3.1 Grid discretization

Instead of symbolic integration, the code evaluates the belief density on a fixed grid of theta values, then normalizes to discrete weights.

```
theta_grid: array of length G
w_prior[i]   ∝ pdf_prior(theta_grid[i])
w_signal[i]  ∝ pdf_signal(theta_grid[i])
w_blended    = update(prior, signal, credibility)   # parametric or linear pool
Normalize all weights so they sum to 1.
```

3.2 Why log-space + a tiny floor exists

At extreme tails, continuous PDFs can be so small that floating-point arithmetic rounds them down to exactly 0. If a grid weight becomes 0, that theta scenario is literally impossible in the discrete approximation, which can look like hard truncation to a reviewer.

To avoid that, the implementation evaluates densities in log space (stable) and optionally enforces a vanishingly small positive floor on weights (e.g., 1e-300) before normalization. The floor is small enough to be scientifically negligible but prevents machine-zero truncation.

4. What gets stored (full trace + exact dict schema)

Every model run produces a single Python dictionary (the result bundle). You save it as result.pkl.gz. This file is authoritative: it contains all flows, times, utilities, probabilities, beliefs, and per-iteration traces.

A small companion file result_summary.json is also written for quick inspection. It converts tuple keys like (o,d) to strings like 'o->d' to satisfy JSON rules.

4.1 Top-level schema (result bundle)

```
result = {
  "settings": {...},
  "reference": {...},

  "beliefs": {
      "prior":   {...},
      "signal":  {...},
      "blended": {...},
      "pooling_method": "parametric" | "linear_pool",
      "credibility": float,
      "market_penetration": float,
  },
  "beliefs_discrete": {
      "theta_grid": np.ndarray(G,),
      "prior":   {"weights": np.ndarray(G,), "logpdf": np.ndarray(G,), "config": {...}},
      "signal":  {"weights": np.ndarray(G,), "logpdf": np.ndarray(G,), "config": {...}},
      "blended": {"weights": np.ndarray(G,), "logpdf": np.ndarray(G,), "config": {...}},
  },

  "equilibrium": {
      "qU_by_od": dict[(o,d)] -> np.ndarray(K,),
      "qI_by_od": dict[(o,d)] -> np.ndarray(K,),
      "q_mix_by_od": dict[(o,d)] -> np.ndarray(K,),
      "od_demands_real": dict[(o,d)] -> float,

      "final_path_flows_total_by_od": dict[(o,d)] -> np.ndarray(K,),
      "final_path_flows_U_by_od":     dict[(o,d)] -> np.ndarray(K,),
      "final_path_flows_I_by_od":     dict[(o,d)] -> np.ndarray(K,),

      "final_state_total": {...state...},
      "final_state_uninformed": {...state...},
      "final_state_informed": {...state...},
      "iterations": int,
  },

  "benchmarks": {
      "UE": {...state...},
      "SO": {...state...},
  },

  "utilities": {
      "final_utility_U_by_od": dict[(o,d)] -> np.ndarray(K,),
      "final_utility_I_by_od": dict[(o,d)] -> np.ndarray(K,),
  },

  "convergence": {"converged": bool, "final_change": float},

  "iteration_history": [ ... ],  # per-iteration records (if enabled)
  "theta_diagnostics":  {...},   # optional per-iteration per-theta diagnostics
```

```
}
```

4.2 State schema (UE / SO / BE total / BE groups)

```
state = {
  "link_flows":  pandas.Series indexed by MultiIndex(u,v),
  "link_times":  pandas.Series indexed by MultiIndex(u,v),
  "path_flows_by_od": dict[(o,d)] -> np.ndarray(K,),
  "path_times_by_od": dict[(o,d)] -> np.ndarray(K,),
  "TSTT": float,

  # UE/SO states also add:
  "iterations": int,
  "converged": bool,
  "final_change": float,
  "od_demands": dict[(o,d)] -> float
}
```

5. What is a path flow? And how are group flows split?

A path probability is 'what fraction of an OD's travelers choose path k.' A path flow is 'how many trips that corresponds to.'

```
For a given OD pair:
  demand = D_od           (trips)
  probability on path k = q_od[k]   (unitless, sums to 1)
  path flow on k = f_od[k] = D_od * q_od[k]   (trips)
```

In BE you have two probability vectors (uninformed vs informed). Market penetration m splits the population:

```
q_mix = (1-m) * q_U + m * q_I

Uninformed path flows: f_U = (1-m) * D_od * q_U
Informed path flows:   f_I = m     * D_od * q_I
Total path flows:      f_T = f_U + f_I
```

The solver stores all three: total, uninformed-only, and informed-only path flows, plus the corresponding link flows and TSTT.

6. How to extract every metric (recipes)

All serious analysis should load the pickle bundle (result.pkl.gz). JSON is only a quick preview because JSON cannot preserve tuple keys, numpy arrays, or pandas objects cleanly.

6.1 Load a single model run

```
import gzip, pickle

with gzip.open(r"...\model_dir\result.pkl.gz", "rb") as f:
    res = pickle.load(f)

be_total = res["equilibrium"]["final_state_total"]
ue       = res["benchmarks"]["UE"]
so       = res["benchmarks"]["SO"]

print("BE TSTT:", be_total["TSTT"])
print("UE TSTT:", ue["TSTT"])
print("SO TSTT:", so["TSTT"])
```

6.2 Extract link flows and link times

```
link_flows = be_total["link_flows"]   # pandas.Series indexed by (u,v)
link_times = be_total["link_times"]

# Tidy DataFrame (good for CSV export)
df_links = (
    link_flows.rename("flow")
    .to_frame()
    .join(link_times.rename("time"))
    .reset_index()
)

df_links.to_csv("be_link_flows_times.csv", index=False)
```

6.3 Extract path flows and path times (per OD)

```
pf_by_od = be_total["path_flows_by_od"]    # dict[(o,d)] -> np.array(K,)
pt_by_od = be_total["path_times_by_od"]    # dict[(o,d)] -> np.array(K,)

# Example: one OD
od = (1, 20)
print("flows:", pf_by_od[od])
print("times:", pt_by_od[od])

# Flatten to a tidy table
rows = []
for (o,d), flows in pf_by_od.items():
    times = pt_by_od[(o,d)]
    for k, (f, t) in enumerate(zip(flows, times), start=1):
        rows.append({"o": o, "d": d, "k": k, "path_flow": float(f), "path_time": float(t)})

import pandas as pd
df_paths = pd.DataFrame(rows)
df_paths.to_csv("be_path_table.csv", index=False)
```

6.4 Extract group split flows

```
be_U = res["equilibrium"]["final_state_uninformed"]
be_I = res["equilibrium"]["final_state_informed"]
```

```
print("Uninformed TSTT:", be_U["TSTT"])
print("Informed TSTT:",   be_I["TSTT"])
```

7. Make a BE vs UE vs SO TSTT comparison table

This creates one row per model run with explicit BE/UE/SO TSTT columns and deltas.

```
import os, json, gzip, pickle
import pandas as pd

out_root = r"C:\Users\Devin\...\Attempted Full Runs\outputs"   # <-- change

rows = []
for name in sorted(os.listdir(out_root)):
    model_dir = os.path.join(out_root, name)
    if not os.path.isdir(model_dir):
        continue

    pkl = os.path.join(model_dir, "result.pkl.gz")
    meta_path = os.path.join(model_dir, "model_meta.json")
    if not os.path.exists(pkl):
        continue

    with gzip.open(pkl, "rb") as f:
        res = pickle.load(f)

    meta = {}
    if os.path.exists(meta_path):
        with open(meta_path, "r") as f:
            meta = json.load(f)

    be = res["equilibrium"]["final_state_total"]["TSTT"]
    ue = res["benchmarks"]["UE"]["TSTT"]
    so = res["benchmarks"]["SO"]["TSTT"]

    rows.append({
        "model_dir": name,
        **meta,
        "TSTT_BE": float(be),
        "TSTT_UE": float(ue),
        "TSTT_SO": float(so),
        "BE_minus_UE": float(be - ue),
        "BE_minus_SO": float(be - so),
        "BE_over_UE_pct": float((be/ue - 1.0) * 100.0),
        "BE_over_SO_pct": float((be/so - 1.0) * 100.0),
    })

df = pd.DataFrame(rows).sort_values("TSTT_BE")
df.to_csv("tstt_comparison.csv", index=False)

# Optional: add a clean 'BE vs UE vs SO' comparison column
df["winner_TSTT"] = df[["TSTT_BE","TSTT_UE","TSTT_SO"]].idxmin(axis=1)
```

That last line 'winner_TSTT' is a simple way to label which of BE/UE/SO has the smallest TSTT per run (useful for quick plots).

8. Troubleshooting (common issues)

8.1 Overflow warnings in exp() (ARA/RRA)

Earlier versions could overflow when computing exponential utilities for large travel times. In v5.4, the exponential argument is clipped to a safe range (about +/-700) so exp() cannot overflow, while preserving monotonicity. This is separate from (and does not touch) the PDF-floor logic for beliefs.

8.2 JSON save errors about tuple keys

If you see TypeError: keys must be str (not tuple), it means you tried to dump a dict with tuple keys into JSON. v5.4's JSON sanitizer converts tuple keys into stable strings (e.g., '1->20'), so result_summary.json can always be written. The pickle bundle never had this limitation.

8.3 K too small

If K is too small (or K-paths are missing for an OD), UE/SO/BE are being computed on a restricted path set that may exclude important alternatives. For publication-quality runs, consider increasing K and verifying path coverage.

Appendix A - Minimal runner (CSV Sioux Falls) using v5.4

This appendix is a skeleton that matches the v5.4 filenames used throughout this guide. It shows the exact inputs you need to provide and the call you make into the solver.

```
# run_sioux_falls_all_models_from_csv_v4.py  (skeleton)
import os, json
import helper_scripts_v5_4_networkwide_trace as hs

CSV_DIR = r"C:\Users\Devin\OneDrive - University of Connecticut\Info Design\Sioux Falls Attempt\Transport
OUT_DIR = r"C:\Users\Devin\OneDrive - University of Connecticut\Info Design\Sioux Falls Attempt\Attempted

# 1) Load CSVs -> build network data (use the same parsing logic as your v3 runner)

# 2) Define prior/signal theta beliefs
prior  = {"dist": "uniform", "low_mult": 0.8, "high_mult": 1.2}
signal = {"dist": "beta",    "low_mult": 0.9, "high_mult": 1.1, "a": 2.0, "b": 5.0}

# 3) Run BE:
res = hs.find_mixed_strategy_equilibrium_multi_od(
    net=net,
    demand_base_by_od=demand_base_by_od,
    theta_real=1.0,
    prior=prior,
    signal=signal,
    credibility=0.75,
    market_penetration=0.50,
    utility_base="prospect",        # or "eut"
    risk_model="variance",          # "ara" or "rra"
    reference_method="UE",          # "UE", "behavioral", or "t_ff_scaled"
    max_iters=60,
    tol=1e-6,
    store_iter_history=True,
)

# 4) Save:
hs.save_result_bundle(res, out_dir=OUT_DIR, base_name="result")
```