Here are several complementary approaches to uncover discriminative sequential patterns (longer than simple cycles) that occur more often in dissatisfied sessions than in satisfied ones.

---

1. Sliding-window *k*-gram counts

The simplest approach is to treat each session's page sequence as a "string" and extract all contiguous subsequences (k-grams) up to some maximum length *K*. Then you count support in each group and rank by the difference (or lift).

Pipeline:

1. **Choose** a maximum pattern length *K* (e.g. 6–8).

For each session sequence (a list of page IDs):

```
 from collections import Counter, defaultdict

def extract_kgrams(seq, k):
    return [tuple(seq[i:i+k]) for i in range(len(seq)-k+1)]

# initialize counters
sup_diss = [Counter() for _ in range(K+1)]
sup_sat  = [Counter() for _ in range(K+1)]

# iterate sessions
for seq, label in data:
    for k in range(2, K+1):              # start at 2 for length ≥2
        kgrams = extract_kgrams(seq, k)
        if label == 0:
            sup_diss[k].update(kgrams)
        else:
            sup_sat[k].update(kgrams)
```

2.
3. **Compute** for each pattern *p* of length *k*:
   $\Delta(p) = \frac{\mathrm{support}_{\rm diss}(p)}{N_{\rm diss}} \;-\; \frac{\mathrm{support}_{\rm sat}(p)}{N_{\rm sat}}$
4. **Rank** patterns by Δ(p), and optionally filter by a minimum support in the dissatisfied group (e.g. appears in ≥1 % of diss sessions).

This approach is embarrassingly parallel and runs in O(N · K) time, with very modest memory if you prune low-support patterns at each *k*.

---

2. Constrained PrefixSpan / SPAM

You mentioned `prefixspan` was too slow on the full data. Two tweaks make it practical:

- **Maxlen constraint**: limit the algorithm to patterns of length ≤ *K*.
- **Minimum support**: don't try to mine extremely rare patterns.
- **Early filtering**: mine only in the dissatisfied subset, then check support in the satisfied set.

For example, using PyPI's `prefixspan`:

```
from prefixspan import PrefixSpan

# mine only on dissatisfied sessions
diss_seqs = [seq for seq,label in data if label==0]

ps = PrefixSpan(diss_seqs)
ps.minlen = 2
ps.maxlen = K
ps.min_support = int(0.01 * len(diss_seqs))  # e.g. 1% support
patterns = ps.frequent(threshold=ps.min_support)

# now for each (pattern, supp_diss), look up supp_sat:
sat_counter = Counter(tuple(seq[i:i+len(pat)])
                      for seq,label in data if label==1
                      for i in range(len(seq)-len(pat)+1))
```

This drastically cuts down the search space.

---

3. Emerging Pattern Mining

Rather than mining all frequent patterns, you can directly search for **emerging patterns**—those whose support significantly increases between two datasets.

- Define support ratio
  ER(p)=suppdiss(p)suppsat(p)+ϵ\mathrm{ER}(p)=\dfrac{\mathrm{supp}_{\rm diss}(p)}{\mathrm{supp}_{\rm sat}(p)+\epsilon}.

- Mine with algorithms like **DEEP** or **CSPADE** (e.g. via the SPMF Java library, which you can call from Python).

---

4. Markov-chain transition anomalies

If you'd like to focus on **transition** patterns rather than arbitrary subsequences:

1. Build first-order (or higher) transition matrices for each user:
   Tij(u)=# (i→j) in user u∑k# (i→k) T_{ij}^{(u)} = \frac{\#\,(i\to j)\ \text{in user }u}{\sum_k \#\,(i\to k)}
2. Aggregate across users in each group to get TdissT^{\rm diss} and TsatT^{\rm sat}.

3. Compute differences ΔT=Tdiss−Tsat\Delta T = T^{\rm diss} - T^{\rm sat} and look for state-pairs (i→j) with large positive Δ.

You can extend this to *n*-step transitions (e.g. bi-grams in the Markov chain) to capture longer navigation paths.

---

5. Sequence embedding + clustering

1. **Embed** each session sequence into a fixed-length vector (e.g. Doc2Vec, RNN autoencoder, or transformer-based encoder).

2. **Cluster** embeddings (k-means, hierarchical).

3. **Label** clusters by the proportion of dissatisfied sessions.

4. **Inspect** the most "dissatisfied" clusters' centroid sequences for common subsequences.

This approach often surfaces non-contiguous patterns (not just strict k-grams).

Sample Python snippet: *k*-gram differential mining

```
from collections import Counter

def find_top_kgram_diffs(data, K=6, min_count=50, top_n=50):
    # data = list of (seq, label)
    N_diss = sum(1 for _,l in data if l==0)
    N_sat  = len(data) - N_diss

    sup_diss = [Counter() for _ in range(K+1)]
    sup_sat  = [Counter() for _ in range(K+1)]

    for seq, label in data:
        for k in range(2, K+1):
            for i in range(len(seq)-k+1):
                pat = tuple(seq[i:i+k])
                (sup_diss if label==0 else sup_sat)[k][pat] += 1

    diffs = []
    for k in range(2, K+1):
        for pat, cnt_d in sup_diss[k].items():
            if cnt_d < min_count:
                continue
            cnt_s = sup_sat[k].get(pat, 0)
            score = cnt_d/N_diss - cnt_s/N_sat
            diffs.append((score, cnt_d, cnt_s, pat))
    # get top by score
    return sorted(diffs, reverse=True)[:top_n]

# Usage
top_patterns = find_top_kgram_diffs(data, K=8, min_count=100, top_n=20)
for score, cd, cs, pat in top_patterns:
    print(f"{pat}  diss:{cd}  sat:{cs}  diff:{score:.3f}")
```

Next steps

1. **Pick one** or combine multiple methods above.

2. **Tune** parameters: max length *K*, min support threshold, and which measure (absolute diff vs. lift).

3. **Validate** patterns by manual inspection or small-scale A/B tests in a hold-out set.

4. **Visualize** the top sequences (e.g. sankey diagrams or heatmaps of ΔT) to communicate the insights.

Once your sessions are represented as NetworkX DiGraph objects, you can mine *graph-based* patterns (not just cycles) and compare their prevalence across groups. Here are two concrete strategies:

---

1. Enumerate and count simple paths (walk-based patterns)

Instead of just cycles, you can extract **all simple paths** up to a given length *L* from each session's graph, canonicalize them, and then compare supports just like k-grams.

```
import networkx as nx

from collections import Counter


def canonical_path(path):

    # tuple form is already canonical for a directed walk

    return tuple(path)


def extract_paths(G, max_len):

    paths = []

    for source in G.nodes():

        # all simple paths starting at source of length ≤ max_len

        for target in G.nodes():

            if source==target: continue

            for path in nx.all_simple_paths(G, source, target, cutoff=max_len):
```

```
                paths.append(canonical_path(path))
    return paths

# Counters for each group
path_counts = {
    0: Counter(),   # dissatisfied
    1: Counter(),   # satisfied
}

for G, label in session_graphs:   # session_graphs = list of (DiGraph, satisfaction)
    for p in extract_paths(G, max_len=5):
        path_counts[label][p] += 1

# Now compute difference in normalized support
N = {l: sum(path_counts[l].values()) for l in (0,1)}
diffs = []
for pat, cnt_d in path_counts[0].items():
    cnt_s = path_counts[1].get(pat, 0)
    score = cnt_d/N[0] - cnt_s/N[1]
    if cnt_d >= 50:   # minimum raw count filter
        diffs.append((score, cnt_d, cnt_s, pat))

# Top patterns more prevalent in dissatisfied
top = sorted(diffs, reverse=True)[:20]
```

```
for score, cd, cs, pat in top:
    print(f"{pat}  diss:{cd}  sat:{cs}  Δ={score:.3f}")
```

- **Pros**: captures *non-cyclic* navigational motifs of any shape (paths).
- **Tuning**: vary `max_len` and the raw-count threshold.

---

2. Frequent subgraph mining

If you want *any* directed subgraph (not only simple paths), you can use subgraph-isomorphism tools to mine **frequent subgraphs**:

1. **gSpan** (via python implementation like `pygspan`) or
2. **NetworkX**'s `DiGraphMatcher` + your own enumeration of small template graphs (e.g. all 3-node directed motifs).

Example: Enumerate all 3-node directed motifs

```
import itertools
from networkx.algorithms import isomorphism

# generate all possible 3-node directed graphs (up to isomorphism)
templates = []
for edges in itertools.product([0,1], repeat=3*2):  # for each ordered pair among 3 nodes
    Gt = nx.DiGraph()
    Gt.add_nodes_from([0,1,2])
    idx = 0
    for u in [0,1,2]:
```

```
        for v in [0,1,2]:
            if u!=v and edges[idx]:
                Gt.add_edge(u,v)
            idx+=1
    # dedupe by isomorphism
    if not any(nx.is_isomorphic(Gt, H) for H in templates):
        templates.append(Gt)

# count each motif in each session
motif_counts = {0: Counter(), 1: Counter()}
for G, label in session_graphs:
    for i,Tpl in enumerate(templates):
        matcher = isomorphism.DiGraphMatcher(G, Tpl)
        # count distinct subgraph isomorphisms
        motif_counts[label][i] += sum(1 for _ in matcher.subgraph_isomorphisms_iter())

# compute emerging motifs
emerging = []
for i in range(len(templates)):
    cd = motif_counts[0][i]
    cs = motif_counts[1][i]
    score = cd - cs
    if cd > 30:
        emerging.append((score, cd, cs, i))
```

```
emerging.sort(reverse=True)

for score, cd, cs, i in emerging[:10]:

    print(f"Motif#{i}  diss:{cd}  sat:{cs}  Δ={score}")
```

- **Pros**: finds *structural* motifs beyond linear walks—e.g. “feed-forward loops,” “bi-fan” patterns, etc.
- **Cons**: combinatorial blow-up as you increase motif size; typically works for 3–5 nodes.

---

3. Aggregated-graph Δ-analysis

Finally, you can build **group-level aggregate graphs**—one for dissatisfied, one for satisfied—where each directed edge $(u \to v)$(u\!\to\!v) has weight = total count of that transition across all sessions in that group:

```
from collections import defaultdict


agg = {0: defaultdict(int), 1: defaultdict(int)}

for G,label in session_graphs:

    for u,v in G.edges():

        agg[label][(u,v)] += 1


# Compute Δ-edge-weights

edge_deltas = []

for edge, w_d in agg[0].items():

    w_s = agg[1].get(edge,0)

    edge_deltas.append((w_d - w_s, w_d, w_s, edge))

edge_deltas.sort(reverse=True)
```

```
for delta, wd, ws, (u,v) in edge_deltas[:20]:
    print(f"{u}→{v}  diss:{wd}  sat:{ws}  Δ={delta}")
```

You can then visualize the edges with large Δ on the dissatisfied aggregate graph to highlight “pain-point” transitions.

Choosing the right approach

- **Paths** (method 1) are easiest and directly generalize your cycle code.
- **Motifs / subgraphs** (method 2) capture richer structures but cost more compute.
- **Aggregate Δ-graphs** (method 3) give you a high-level map of transition hotspots.

Adding time deltas opens up a whole range of richer, time-aware pattern analyses. Here are several concrete ways you can fold “dwell time” into your graph-based analysis:

---

1. Weighted transition graphs

Rather than a plain DiGraph where each edge is just a count of transitions, store your time deltas as edge attributes:

```
import networkx as nx

from collections import defaultdict



# aggregate per group

agg_time = {0: defaultdict(list), 1: defaultdict(list)}

for seq, deltas, label in data:

    # seq = [“Home”, “About”, …], deltas = [2,3,…]

    for u, v, t in zip(seq, seq[1:], deltas):

        agg_time[label][(u,v)].append(t)



# build weighted graphs

graphs = {}

for label in (0,1):

    G = nx.DiGraph()

    for (u,v), times in agg_time[label].items():

        G.add_edge(u, v,

                   count=len(times),

                   avg_time=sum(times)/len(times),

                   median_time=sorted(times)[len(times)//2])
```

```
    graphs[label] = G
```

You can then compare:

- **Avg dwell per transition** Δ = avg_time_diss − avg_time_sat
- **Volume-weighted time** (count × avg_time) differences to highlight “slow, repeated” transitions.

---

2. Time-constrained path mining

When extracting simple paths, you can enforce **total** or **per-edge** time constraints:

```
def extract_time_constrained_paths(G, max_len, min_total_time=None):
    valid = []
    for source in G.nodes():
        for target in G.nodes():
            if source==target: continue
            for path in nx.all_simple_paths(G, source, target, cutoff=max_len):
                # sum the dwell times on each edge
                total = sum(G[u][v]['avg_time'] for u,v in zip(path, path[1:]))
                if min_total_time is None or total >= min_total_time:
                    valid.append((tuple(path), total))
    return valid


# e.g. find paths of ≤5 hops that spend ≥15s total
paths = extract_time_constrained_paths(graphs[0], max_len=5, min_total_time=15)
```

Comparing these time-filtered path counts between groups can surface “long but frequent” frustrated journeys.

---

3. Temporal motif mining

Small directed subgraphs (motifs) might carry different **temporal signatures**. For each 3-node motif you detect, also capture the **distribution** of time deltas on the involved edges. Then compare motif **frequency × duration** between groups.

---

4. Process-mining / directly-follows graphs

Treat your data as an **event log**: each page view is an event `(case_id, activity, timestamp)`. Tools like **PM4Py** let you:

1. Discover a **directly-follows graph** annotated with performance (avg transition times).
2. Identify the most common **variants** (full sequences) separately for satisfied vs. dissatisfied.
3. Compare the **performance metrics** (e.g. throughput times) on those variants.

This is essentially graph mining plus time-performance analysis, and is very effective at surfacing “pain-point” sub-processes.

---

5. Feature vector + clustering / classification

You can also reduce each session to a feature vector of:

- Counts of your top N time-aware paths or motifs
- Summary stats (mean, median, max dwell, total time)
- Graph metrics (e.g. avg out-degree weighted by time)

Then train a simple classifier (e.g. logistic regression) to pick the most discriminative features—those with highest coefficients will point to time-aware patterns that predict dissatisfaction.

Next steps

1. **Decide** which perspective you want—edge-level times (method 1), path filtering (method 2), motif durations (method 3), or full process models (method 4).

2. **Prototype** on a small subset to gauge performance.

3. **Visualize** key insights: e.g. a heat-map of Δ avg_time on your aggregate graph or sankey diagram of slowest—most divergent—paths.