# Bidirectional Pathfinding Experiment Report

טל קליין – 209234103, יובל מירון – 209044148

## Introduction

This project investigates pathfinding algorithms for pedestrian navigation around Ben-Gurion University (BGU) utilizing OpenStreetMap (OSM) data. The principal algorithm employed is Bidirectional A* search, which operates by initiating two concurrent A* searches, one progressing from the origin and the other from the destination, converging at an intermediate point to enhance efficiency. This method employs admissible heuristics, specifically straight-line distance, to ensure optimal routing while substantially decreasing both the search space and computation time relative to single-directional A* search.

The developed pathfinding system accommodates a range of routing objectives (cost functions), extending beyond shortest-distance calculations to include minimization of traffic light stops, maximization of safety, and reduction of travel time. These varied optimization criteria provide a comprehensive framework for analyzing the impact of each factor on route selection.

## Experiment Goal

The experiment is structured to evaluate the performance of a Bidirectional A* pathfinding algorithm across different route optimization objectives. The main hypothesis is that bidirectional search will effectively identify optimal paths for each specified criterion, and that each routing objective will result in distinct route features consistent with its optimization target. Specifically:

- The Shortest Distance algorithm generates routes with the minimum walking distance, which may involve more road crossings or traffic lights.
- The Fewest Traffic Lights algorithm produces routes with fewer stops at traffic signals or crossings than the shortest distance path, possibly resulting in longer distances or travel times.
- The Safest Route algorithm finds paths with lower safety penalties—such as fewer unlit or higher-risk segments, compared to the shortest route, potentially increasing total distance or time due to detours through safer areas.

- The Fastest Time algorithm identifies the quickest routes in terms of travel time, which may not correspond to the shortest distance; for instance, it can avoid congested intersections or slow segments to achieve reduced overall travel time.

Overall, the experiment assesses whether each custom cost function (distance, delays, safety, time) reliably directs the A* search toward an optimal path reflecting the intended priority and quantifies the trade-offs associated with each objective, such as additional distance traveled to minimize signal stops or enhance safety.

## Experiment Description

Software & Data: The experiment was conducted using Python and a custom OpenStreetMap (OSM) pathfinding system. OSM map data (https://github.com/TalKleinBgu/osm-pathfinder-bgu/blob/main/map.osm) for the Ben-Gurion University campus and adjacent Beer-Sheva neighborhoods was parsed into a pedestrian graph tailored to walking. During parsing, all OSM nodes (with coordinates and tags like traffic signals and lighting) and ways (ordered node lists with tags) are loaded. A walkability filter excludes car-only roads and only includes pedestrian-suitable paths like footways, sidewalks, and residential streets.

The designated destination was a specific campus student house (בית הסטודנט), identified by its OSM node ID. Three student neighborhoods in Beer-Sheva—Neighborhood B (שכונה ב), Neighborhood D (שכונה ד), and Old V (ו הישנה), were selected as source areas. For each neighborhood, the locations of all buildings were determined to serve as starting points for routing. In total, approximately 4,724 building-to-campus routes were calculated (1,181 building * 4 path = 4724 path).

Before searching, both endpoints are snapped to the walking graph. For OSM nodes or building ways, the nearest walkable node is found by straight-line distance. building sources use the centroid, also snapped to the closest graph node. This approach keeps searches on graph vertices and avoids mid-segment geometry interpolation.

Algorithm Implementation: Routing uses a Bidirectional A*. Two A* searches run simultaneously: the forward search from the start and the backward search from the goal using the reverse graph. Each maintains its own open queue keyed by $f(n)=g(n)+h(n)$ and a closed set to prevent re-expansion. The implementation keeps *g-scores* (gF, gB) and predecessor maps (predF, predB) on both sides. A priority-

queue routine removes entries whose g-score doesn't match the current best score. At every step the side with the smaller current f expands next. After expanding a node, the algorithm (i) tightens an upper bound (UB) whenever it discovers a vertex already reached by the opposite search, and (ii) records a meeting state that may be either a node (both searches reached the same vertex) or an edge hand-off (forward reached u and backward reached v with an admissible connection across u→v). The global termination rule follows the standard bidirectional A* invariant:

$$\min_{x \in \mathrm{Open_F}} f_F(x) + \min_{y \in \mathrm{Open_B}} + f_B(y) \geq UB$$

Correctness and optimality are enforced by admissible heuristics and non-negative edge costs across all profiles, with each profile pairing a suitable heuristic to its cost model and keeping the reverse search symmetric so a front-to-front meet remains optimal. For *shortest* ,the heuristic is the Haversine distance, which never overestimates, for *fastest*, it is Haversine distance divided by a generous upper-bound walking speed (1.6 m/s), ensuring actual segment speeds never exceed the bound, for *few_traffic_lights*, the heuristic scales Haversine distance by distance factor then applies a 0.95 safety margin to stay strictly optimistic even when penalties for entering signal or crossing nodes appear, for *safest*, the heuristic remains Haversine distance while the edge cost starts at that distance and only adds non-negative safety penalties (lighting, sidewalks, road class, surface, narrowness, crossings), so the adjusted cost is never below the base distance, preserving admissibility.

Execution Details: The experiment was run end-to-end on a standard desktop using Python. For each area of interest (the three Be'er-Sheva neighborhoods), the pipeline loads the OSM map, parses nodes/ways, and builds a pedestrian graph in both directions (forward and reverse). Each building is then snapped to its nearest walkable graph vertex, the fixed destination (the BGU Student house, OSM id 135310103) is likewise snapped to the closest walkable node. From that start–goal pair the system executes four independent searches—shortest, few_traffic_lights, safest and fastest—each via proper Bidirectional A* with the profile's admissible heuristic. Because the graph window is compact (≈2 km around campus) and bidirectional search halves effective depth, queries complete on the millisecond scale per search. During search the solver records diagnostics (forward/backward node expansions, edges scanned) and, after reconstructing the path, computes uniform route metrics over the final node sequence: geometric distance, time, counts of traffic signals and crossings, cumulative safety penalties, detour factor (vs. straight-line), and realized average walking speed.

All source code, scripts, and raw data files used for this experiment are publicly available in GitHub repository - https://github.com/TalKleinBgu/osm-pathfinder-bgu

Additionally, interactive map visualizations and all computed paths can be explored through the following links:

- https://talkleinbgu.github.io/osm-pathfinder-bgu/NeighborhoodB.html – displays all computed paths from Neighborhood B to the campus

- https://talkleinbgu.github.io/osm-pathfinder-bgu/NeighborhoodD.html – shows optimal routes from Neighborhood D

- https://talkleinbgu.github.io/osm-pathfinder-bgu/NeighborhoodOldV.html – presents the route visualizations for Old V neighborhood

## Results

Despite all routes sharing the same destination, the outcomes demonstrate clear differences in path choices and metrics for the four algorithms, confirming the experiment's expectations.

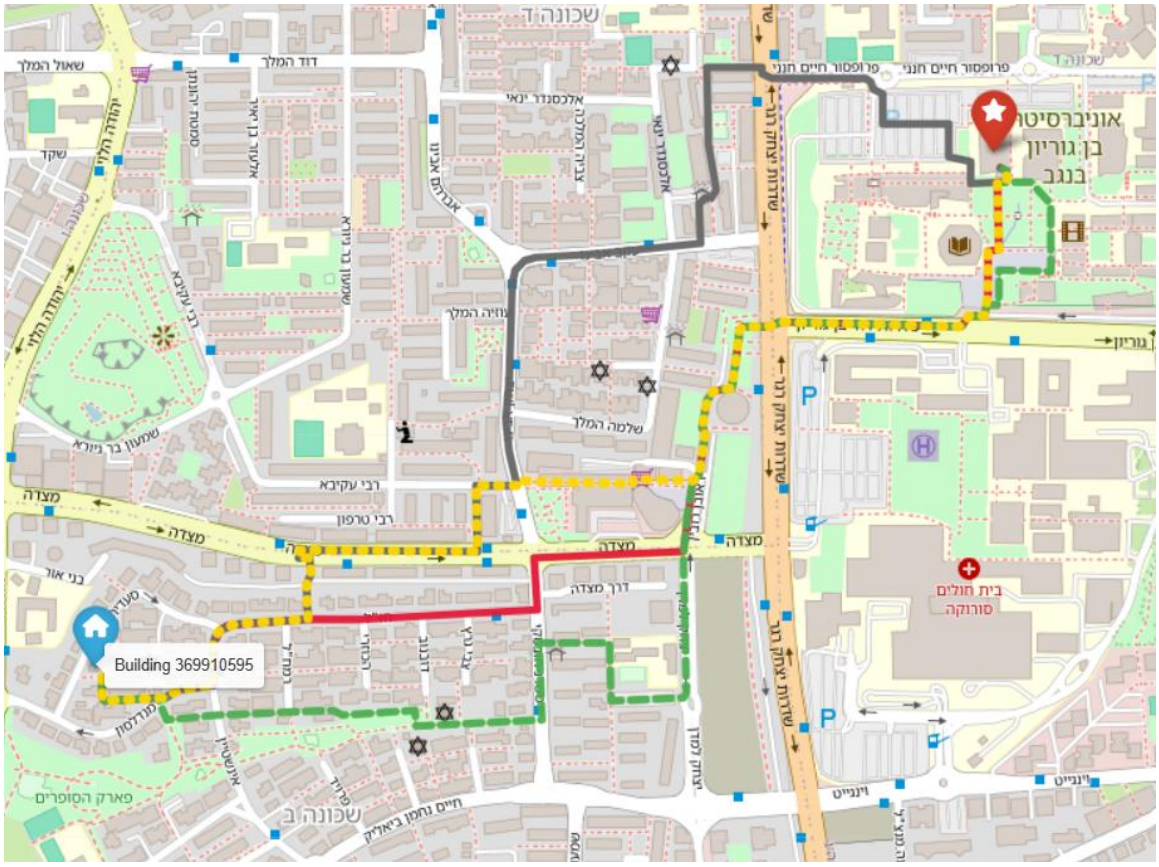
All statistics for every buildings are located in the link - https://github.com/TalKleinBgu/osm-pathfinder-bgu/blob/main/neighborhood_pathfinding_detailed_data.csv

The next table summarizes an example set of results for a representative building (369910595) under each routing profile:

| Algorithm | Path Distance (m) | Travel Time (min) | Traffic Signals | Detour factor | Num nodes in path | Avg walking speed | Signal density per 100km | Edge scanned | Node expanded |
|---|---|---|---|---|---|---|---|---|---|
| Shortest (distance) | 1787 | 25.07 | 6 | 1.33 | 80 | 4.28 | 0.34 | 1357 | 536 |
| Fewest Traffic Lights | 2152 | 27.76 | 2 | 1.6 | 114 | 4.65 | 0.09 | 7539 | 3257 |
| Safest Route | 2178 | 28.28 | 5 | 1.62 | 76 | 4.62 | 0.23 | 2962 | 1213 |
| Fastest Time | 1852 | 24.5 | 3 | 1.38 | 86 | 4.53 | 0.16 | 2395 | 981 |

* Detour factor is (route distance ÷ straight-line (haversine) distance); 1.0 is perfectly direct, 1.3 means the walk is 30% longer than the haversine.

The Shortest profile provides the geometric baseline (1,787 m in 25.07 min) but traverses 6 signals, which depresses realized speed (4.28 km/h). Optimizing for Fewest Traffic Lights does exactly what it promises—signals drop to 2 and signal density plunges from 0.34 to 0.09 per 100 km, but at a notable cost: distance grows by to 2,152 m, time to 27.76 min, and search work inflates substantially (≈3,257 expansions, ~6× baseline). The Safest profile lands in the middle: distance rises to 2,178 m, time to 28.28 min, and signals fall modestly (to 5), with realized speed improving to 4.62 km/h; computational effort is moderate (1213 expansions). Fastest achieves the most practical efficiency: with only a small geometric detour (1,851 m) it halves signal encounters (3 vs. 6) and finishes faster than Shortest, at a modest computational. In short, for this building the profiles trace a clean trade-off surface: *Shortest* minimizes meters, *Fewest-Lights* minimizes interruptions at the price of extra distance, *Fastest* converts targeted signal avoidance into a small but real time win.




The figure shows four alternatives from Building 369910595 (house icon, left) to the BGU student house (red star, right).

The Shortest route (red) goes mostly straight east, then into campus. It crosses the big main road near the university at several traffic lights. So it is shortest in meters, but you have to stop many times.

The Fastest route (yellow) uses much of the same streets as red, but makes two small turns to reach quicker crossings and wider entries to the student building. These small detours reduce long waits, so you arrive sooner with only a little extra distance.

The Fewest Traffic Lights route (dark blue) goes north earlier, follows a longer, smoother street, crosses the main road once, and comes in from the north side of campus. It has almost no stops, but it is the most indirect path.

The Safest route (green) prefers streets with lights and sidewalks, and it uses marked crossings. It passes near community places like the park and nearby schools/playgrounds, where there are people and better lighting. This adds a small detour for a more comfortable walk.

All four routes meet near the same campus entrance. The main differences come from how each one handles the busy, traffic-light area in the middle of the trip.

When we aggregate by neighborhood, the same patterns hold while the magnitude of the trade-offs depends on local street morphology.

| Neighborhood | Algorithm | Path Distance (m) | Travel Time (min) | Traffic Signals | Detour factor | Num nodes in path | Avg walking speed | Signal density per 100km | Edge scanned | Node expanded |
|---|---|---|---|---|---|---|---|---|---|---|
| B | Shortest (distance) | 1677.68 | 23.2 | 5.04 | 1.37 | 80.71 | 4.33 | 0.31 | 907.91 | 2250.01 |
| | Fewest Traffic Lights | 1919.29 | 25.39 | 3.16 | 1.57 | 89.16 | 4.52 | 0.18 | 3010.17 | 7034.16 |
| | Safest Route | 1979.2 | 26.1 | 5.11 | 1.61 | 84.26 | 4.53 | 0.27 | 1203.96 | 2955.98 |
| | Fastest Time | 1652.46 | 23.71 | 6.7 | 1.34 | 75.87 | 4.17 | 0.42 | 517.82 | 1312.96 |
| D | Shortest (distance) | 1356.74 | 17.79 | 2.63 | 1.44 | 59.24 | 4.55 | 0.21 | 544.23 | 1397.53 |
| | Fewest Traffic Lights | 1426.14 | 18.58 | 2 | 1.52 | 69.91 | 4.57 | 0.15 | 1576.2 | 3903.07 |
| | Safest Route | 1565.76 | 19.97 | 2.6 | 1.65 | 63.25 | 4.68 | 0.18 | 986.04 | 2500.22 |
| | Fastest Time | 1353.51 | 18.43 | 3.37 | 1.44 | 61.77 | 4.38 | 0.27 | 360.79 | 932.74 |
| Old V | Shortest (distance) | 997.16 | 12.46 | 0.02 | 1.76 | 53.86 | 4.81 | 0 | 342.45 | 876.18 |
| | Fewest Traffic Lights | 996.73 | 12.58 | 0.02 | 1.76 | 52.06 | 4.76 | 0 | 317.27 | 809.79 |
| | Safest Route | 1154.19 | 14.58 | 0.5 | 2.01 | 51.66 | 4.77 | 0.04 | 591.81 | 1500.15 |
| | Fastest Time | 992.01 | 12.63 | 0.24 | 1.75 | 51.69 | 4.72 | 0.02 | 302.16 | 772.9 |

When we aggregate by neighborhood, the same patterns hold while the magnitude of the trade-offs depends on local street morphology. In Neighborhood B (the farthest and most signal-dense), path lengths sit around 1.6–1.9 km and signal exposure is highest, so the objectives separate the most: Fewest-Lights yields the largest detour (detour factor ≈ 1.57) and the heaviest search footprint (≈3,010 edges scanned; ≈7,034 expansions. Neighborhood D is intermediate: distances ~1.35–1.43

km and baseline signal exposure are lower, so all profiles cluster closer to Shortest; Fewest-Lights still trims signal density (≈ 0.21→0.15) but with a smaller detour and moderate extra computation. Old V is the closest (≈1.0–1.1 km) and essentially signal-free (densities ≈0–0.03), so the profiles nearly collapse to the same solution; only Safest accepts a slight detour (≈ 2.01) to honor infrastructure preferences, and search costs are uniformly low.

## Conclusion

This study demonstrates that a properly engineered Bidirectional A*, paired with profile-specific but admissible cost models, can compute thousands of optimal pedestrian routes on real OSM data quickly and reliably. Across 4,724 paths (≈1,181 buildings × 4 profiles), each objective produced routes whose characteristics matched the intended priority: *Shortest* minimized distance but encountered the most regulated junctions; *Fewest Traffic Lights* reliably suppressed stop exposure (and raised realized walking speed) at the cost of longer, more circuitous paths and heavier search; *Safest* favored infrastructure quality—lighting, sidewalks, controlled crossings, accepting moderate detours; and *Fastest* achieved small yet consistent time gains by bypassing a subset of signals with minimal added distance. These trends held at both the single-building and neighborhood scales, confirming the hypotheses about how each cost function steers the search.

The magnitude of those trade-offs depended on local street morphology. Neighborhood B—farthest and most signal-dense—amplified differences: the few-lights profile delivered the largest reduction in signal exposure but demanded the biggest detours and the most computation, while fastest preserved near-baseline geometry and lean search effort. Neighborhood D showed the same ordering with smaller gaps, and Old V—closest and nearly signal-free—caused the profiles to collapse to near-identical routes, with only the safety profile choosing mild detours to honor infrastructure preferences.

Practically, these findings translate into clear guidance. When arrival time is paramount and the grid is moderately regulated, *Fastest* is a sensible default: it trims a few delays without meaningfully lengthening the walk. When continuity (few stops) matters—e.g., with strollers, running, or user preference—*Fewest Lights* is appropriate, with the transparent trade-off of more meters and minutes. Where comfort and perceived safety dominate (nighttime or unfamiliar areas), *Safest* provides infrastructure-aligned paths at moderate overhead. And in low-signal settings, *Shortest* (or *Fastest*) is typically sufficient since all profiles converge.

## Appendix: Key source code

Below are the essential, self-contained snippets and patterns that implement the system's core behavior. Each excerpt is trimmed to the minimum needed to understand structure, flow, and design trade-offs; it matches your repository exactly in naming and logic.

The whole code is in the link - https://github.com/TalKleinBgu/osm-pathfinder-bgu

The results json (with paths) in the link - https://github.com/TalKleinBgu/osm-pathfinder-bgu/tree/main/results

1) Core data model

```python
@dataclass
class Node:
    id: str
    lat: float
    lon: float
    tags: Dict[str, str]

@dataclass
class Way:
    id: str
    nodes: List[str]
    tags: Dict[str, str]

@dataclass
class PathResult:
    path: List[str]
    total_cost: float
    distance_meters: float
    num_traffic_signals: int
    num_crossings: int
    safety_penalties: float
    time_seconds: float
    description: str
    nodes_expanded_forward: int = 0
    nodes_expanded_backward: int = 0
    edges_scanned_forward: int = 0
    edges_scanned_backward: int = 0
    stale_pops_forward: int = 0
    stale_pops_backward: int = 0
```

2) OSM parsing and pedestrian graph build (forward + reverse)

```python
def _build_graph(self):
    print("Building graphs (forward & reverse)…")
    for way in self.ways.values():
        tags = way.tags
        # For pedestrians, most oneway restrictions don't apply
        # Only restrict if explicitly foot=no or specific pedestrian oneway restrictions
        oneway_car = tags.get('oneway') == 'yes'
        highway = tags.get('highway', '')

        # Pedestrians can generally walk both ways on most roads, even if oneway for cars
        # Exception: some highway types where pedestrians should follow car direction
        is_oneway_for_ped = False
        if oneway_car:
            # On these highway types, pedestrians should respect car oneway
            if highway in {'motorway', 'motorway_link', 'trunk', 'trunk_link', 'primary'}:
                is_oneway_for_ped = True
            # If there's explicit foot restriction, respect it
            elif tags.get('foot') == 'no':
                continue  # Skip this way entirely for pedestrians

        for i in range(len(way.nodes) - 1):
            a, b = way.nodes[i], way.nodes[i+1]
            if a not in self.nodes or b not in self.nodes:
                continue
            dist = self._haversine(self.nodes[a], self.nodes[b])

            e_ab = {'to_node': b, 'distance': dist, 'way_id': way.id, 'way_tags': tags}
            e_ba = {'to_node': a, 'distance': dist, 'way_id': way.id, 'way_tags': tags}

            # forward
            self.graph[a].append(e_ab)
            # reverse
            self.rgraph[b].append({'to_node': a, 'distance': dist, 'way_id': way.id, 'way_tags': tags})

            # add opposite direction if not one-way for pedestrians
            if not is_oneway_for_ped:
                self.graph[b].append(e_ba)
                self.rgraph[a].append({'to_node': b, 'distance': dist, 'way_id': way.id, 'way_tags': tags})

    print(f"Graph has {len(self.graph)} forward nodes, {len(self.rgraph)} reverse nodes.")
```

```python
def _is_walkable(self, tags: Dict[str, str]) -> bool:
    highway = tags.get('highway', '')
    if highway in {'motorway', 'motorway_link', 'trunk', 'trunk_link'}:
        return False
    foot = tags.get('foot', '')
    if foot == 'no':
        return False
    if foot == 'yes':
        return True
    return highway in {
        'footway','pedestrian','path','steps','sidewalk',
        'primary','secondary','tertiary','residential',
        'living_street','service','unclassified','track'
    }
```

3) Safety context (public places index) and geometry

```python
def _index_public_places(self, radius_m: float = 60.0) -> None:
    """Mark nodes in the walking graph that are at or near public places
    such as playgrounds, sports fields, schools, places of worship.
    Stores flags in Node.tags: near_public_place=yes and public_place_kind.
    """
    (variable) amenity_whitelist: set[str]
    amenity_whitelist = {
        'school','kindergarten','university','library','community_centre',
        'place_of_worship','synagogue','police','clinic','hospital'
    }
    leisure_whitelist = {'playground','pitch','sports_centre','stadium','park','garden'}

    # Collect public place nodes
    public_node_ids: List[str] = []
    for nid, node in self.nodes.items():
        a = node.tags.get('amenity','')
        l = node.tags.get('leisure','')
        if (a in amenity_whitelist) or (l in leisure_whitelist):
            public_node_ids.append(nid)

    if not public_node_ids:
        return

    # Mark graph nodes that are at/near public places
    for nid in self.graph.keys():
        n = self.nodes[nid]
        # direct tag on this node counts
        if nid in public_node_ids:
            n.tags['near_public_place'] = 'yes'
            n.tags['public_place_dist'] = '0'
            continue
        # otherwise compute proximity to nearest public node (simple scan; area is small)
        mind = float('inf')
        for pid in public_node_ids:
            d = self._haversine(n, self.nodes[pid])
            if d < mind:
                mind = d
                if mind <= radius_m:
                    break
        if mind <= radius_m:
            n.tags['near_public_place'] = 'yes'
            n.tags['public_place_dist'] = f"{mind:.1f}"
```

4) Cost models (four profiles, symmetric and admissible)

```python
def _cost_shortest(self, e, from_id):  # meters
    return e['distance']

def _cost_few_traffic_lights(self, e, from_id):
    # Distance scaled down so signals dominate, with nuanced crossing penalties
    cfg = self._few_traffic_lights_cfg
    c = e['distance'] * cfg['distance_factor']

    # penalties on the node being entered
    nt = self.nodes[e['to_node']].tags
    crossing = nt.get('crossing')  # e.g., traffic_signals, zebra, uncontrolled, island
    is_signal = (nt.get('highway') == 'traffic_signals') or (crossing == 'traffic_signals')

    if is_signal:
        c += cfg['signal_penalty_m']
    elif nt.get('highway') == 'crossing' or crossing:
        # nuanced crossing types
        if crossing == 'zebra' or nt.get('crossing:markings') == 'zebra':
            c += cfg['zebra_penalty_m']
        elif crossing == 'uncontrolled' or crossing == 'no_signals':
            c += cfg['uncontrolled_penalty_m']
        elif crossing == 'island' or crossing == 'refuge_island':
            c += cfg['island_penalty_m']
        else:
            c += cfg['other_crossing_penalty_m']

        # accessibility features slightly reduce perceived penalty
        if nt.get('tactile_paving') == 'yes':
            c += cfg['tactile_bonus_m']
        if nt.get('kerb') == 'lowered':
            c += cfg['lowered_kerb_bonus_m']

    return c
```

```python
def _cost_safest(self, e, from_id, is_night: bool):
    c = e['distance']
    wt = e['way_tags']
    # lighting along way or at node
    way_lit = wt.get('lit')
    if (way_lit == 'no') or (way_lit is None and self.nodes[e['to_node']].tags.get('lit') == 'no'):
        c += 150.0 if is_night else 25.0

    if way_lit == 'yes' or (way_lit is None and self.nodes[e['to_node']].tags.get('lit') == 'yes'):
        c -= 10.0

    # alley/service and road classification when mixing with traffic
    hw = wt.get('highway')
    if hw in {'alley','service'}:
        c += 90.0 if is_night else 25.0
    if hw in {'primary','secondary','tertiary'} and wt.get('sidewalk', '') in {'no','none',''}:
        c += 140.0 if is_night else 50.0

    # sidewalks
    sw = wt.get('sidewalk', '')
    if sw in {'none','no',''} and hw not in {'footway','pedestrian','path'}:
        c += 90.0 if is_night else 25.0

    # shared cycle/foot paths without segregation are a bit less comfortable
    if hw in {'path','footway'} and wt.get('bicycle') in {'yes','designated'} and wt.get('segregated') == 'no':
        c += 30.0 if is_night else 10.0

    # surface: poor surfaces feel less safe or comfortable
    surf = wt.get('surface','')
    if surf in {'gravel','sand','dirt','ground','grass','unpaved'}:
        c += 40.0
    if surf in {'cobblestone','sett'}:
        c += 25.0

    # narrow width
    try:
        width = float(wt.get('width', 'nan'))
        if width and width < 1.5:
            c += 30.0
    except ValueError:
        pass

    # tunnels/bridges at night
    if is_night and (wt.get('tunnel') == 'yes' or wt.get('bridge') == 'yes'):
        c += 50.0

    # node-based crossing safety: uncontrolled crossings are worse; zebra better
    nt = self.nodes[e['to_node']].tags
    if nt.get('highway') == 'crossing' or nt.get('crossing'):
        crossing = nt.get('crossing')
        if crossing == 'uncontrolled' or crossing == 'no_signals':
            c += 60.0
        elif crossing == 'zebra' or nt.get('crossing:markings') == 'zebra':
            c += 10.0  # still a minor risk
        elif crossing == 'island':
            c += 20.0

        # accessibility features reduce perceived risk slightly
        if nt.get('tactile_paving') == 'yes':
            c -= 5.0
        if nt.get('kerb') == 'lowered':
            c -= 5.0

    # public places increase perceived safety; apply a small bonus (reduce penalties)
    # Bonus is larger at night. Keep edge cost non-negative by clamping to base distance.
    bonus = 0.0
    amenity = nt.get('amenity','')
    leisure = nt.get('leisure','')
    if (
        amenity in {'place_of_worship','synagogue','school','kindergarten','university','library','community_centre','police','clinic','hospital','restaurant','cafe'}
        or leisure in {'playground','pitch','sports_centre','stadium','park','garden','fitness_centre','dog_park','swimming_pool'}
        or nt.get('near_public_place') == 'yes'
    ):
        bonus = 40.0 if is_night else 20.0
    if bonus:
        c = max(e['distance'], c - bonus)

    return max(e['distance'], c)
```

```python
def _cost_fastest(self, e, from_id):  # seconds
    dist = e['distance']
    wt = e['way_tags']

    # baseline walking speed (m/s) varies by facility
    hw = wt.get('highway')
    if hw in {'footway','pedestrian','sidewalk'}:
        v = 1.45
    elif hw == 'steps':
        v = 0.8
    else:
        v = 1.35

    # surface adjustments
    surf = wt.get('surface','')
    if surf in {'asphalt','concrete','paved'}:
        pass
    elif surf in {'compacted','fine_gravel'}:
        v *= 0.95
    elif surf in {'gravel','sand','dirt','ground','grass','unpaved'}:
        v *= 0.8
    elif surf in {'cobblestone','sett'}:
        v *= 0.9

    # narrow width slows flow slightly
    try:
        width = float(wt.get('width', 'nan'))
        if width and width < 1.5:
            v *= 0.92
    except ValueError:
        pass

    t = dist / max(v, 0.5)

    # steps/elevator handling
    if hw == 'steps':
        t *= 1.5
    # entering elevator nodes/ways incurs wait time; very fast vertical travel
    nt = self.nodes[e['to_node']].tags
    if nt.get('highway') == 'elevator' or wt.get('highway') == 'elevator':
        t += 20.0  # average wait
        # movement within elevator is negligible for distance scales here

    # waiting at crossings/signals when entering node
    crossing = nt.get('crossing')
    if (nt.get('highway') == 'traffic_signals') or (crossing == 'traffic_signals'):
        t += 25.0
    elif nt.get('highway') == 'crossing' or crossing:
        if crossing == 'zebra' or nt.get('crossing:markings') == 'zebra':
            t += 6.0
        elif crossing == 'uncontrolled' or crossing == 'no_signals':
            t += 3.0
        elif crossing == 'island' or crossing == 'refuge_island':
            t += 8.0
        else:
            t += 6.0

    # shared with bicycles and not segregated can slow slightly
    if hw in {'path','footway'} and wt.get('bicycle') in {'yes','designated'} and wt.get('segregated') == 'no':
        t *= 1.03

    return t
```

Admissible heuristics (paired to profile):

```python
    def _haversine(n1: Node, n2: Node) -> float:
        R = 6371000.0
        lat1, lon1 = math.radians(n1.lat), math.radians(n1.lon)
        lat2, lon2 = math.radians(n2.lat), math.radians(n2.lon)
        dlat, dlon = lat2-lat1, lon2-lon1
        a = math.sin(dlat/2)**2 + math.cos(lat1)*math.cos(lat2)*math.sin(dlon/2)**2
        return 2*R*math.asin(math.sqrt(a))

    def _heuristic_distance(self, nid: str, tid: str) -> float:
        return self._haversine(self.nodes[nid], self.nodes[tid])

    def _heuristic_time(self, nid: str, tid: str) -> float:
        return self._heuristic_distance(nid, tid) / 1.6  # m/s (upper-bound speed)
```

5) Bidirectional A*

```python
def bidir_astar(self, start: str, goal: str, profile: str, is_night=False) -> Optional[PathResult]:
    if start == goal:
        return self._calc_stats([start], profile, is_night)
    if start not in self.graph or goal not in self.graph:
        return None

    # Select edge cost
    def edge_cost(e, u):
        if profile == 'shortest': return self._cost_shortest(e, u)
        if profile == 'few_traffic_lights': return self._cost_few_traffic_lights(e, u)
        if profile == 'safest': return self._cost_safest(e, u, is_night)
        if profile == 'fastest': return self._cost_fastest(e, u)
        return self._cost_shortest(e, u)

    # Heuristics: keep admissible for each profile
    def hF(n):  # forward heuristic to goal
        if profile == 'fastest':
            return self._heuristic_time(n, goal)
        if profile == 'few_traffic_lights':
            return 0.95 * self._few_traffic_lights_cfg['distance_factor'] * self._heuristic_distance(n, goal)
        return self._heuristic_distance(n, goal)

    def hB(n):  # backward heuristic to start
        if profile == 'fastest':
            return self._heuristic_time(n, start)
        if profile == 'few_traffic_lights':
            return 0.95 * self._few_traffic_lights_cfg['distance_factor'] * self._heuristic_distance(n, start)
        return self._heuristic_distance(n, start)

    # State
    gF = {start: 0.0}
    gB = {goal: 0.0}
    predF = {start: None}
    predB = {goal: None}
    openF = [(hF(start), 0.0, start)]  # (f, g, node)
    openB = [(hB(goal), 0.0, goal)]
    closedF = set()
    closedB = set()

    UB = float('inf')
    meet = None  # ('node', m) or ('edge', u, v)
    forward_expanded = 0
    backward_expanded = 0
    forward_edges_scanned = 0
    backward_edges_scanned = 0
```

```
    def top_key(pq):
        # return current min f, skipping stale
        while pq and (pq[0][1] > (gF.get(pq[0][2], float('inf')) if pq is openF else gB.get(pq[0][2], float('inf')))):
            heapq.heappop(pq)
        return pq[0][0] if pq else float('inf')

    def improve_UB_via_node(u):
        nonlocal UB, meet
        if u in gF and u in gB:
            val = gF[u] + gB[u]
            if val < UB:
                UB = val
                meet = ('node', u)

    def relax_dir(u, g_u, graph_dir, g_here, g_other, pred_here, h_here, open_here, is_forward: bool):
        nonlocal UB, meet
        nonlocal forward_edges_scanned, backward_edges_scanned

        for e in graph_dir.get(u, []):
            if is_forward:
                forward_edges_scanned += 1
            else:
                backward_edges_scanned += 1

            v = e['to_node']
            c = edge_cost(e, u)
            if c < 0:
                continue  # safety
            tentative = g_u + c
            if tentative < g_here.get(v, float('inf')):
                g_here[v] = tentative
                pred_here[v] = u
                f = tentative + h_here(v)
                heapq.heappush(open_here, (f, tentative, v))
            # Try to tighten UB using v if the other search has reached v
            if v in g_other:
                cand = tentative + g_other[v]
                if cand < UB:
                    UB = cand
                    meet = ('edge', u, v) if is_forward else ('edge', v, u)  # store as (u->v) in forward direction
```

```
        # Main loop
        while openF or openB:
            # Expand side with smaller current f
            if top_key(openF) <= top_key(openB):
                # Forward step
                f_u, g_u, u = heapq.heappop(openF)
                if u in closedF:
                    continue
                if g_u != gF.get(u, float('inf')):
                    stale_pops_f += 1
                    continue
                closedF.add(u)
                forward_expanded += 1
                improve_UB_via_node(u)
                relax_dir(u, g_u, self.graph, gF, gB, predF, hF, openF, True)
            else:
                # Backward step
                f_u, g_u, u = heapq.heappop(openB)
                if u in closedB:
                    continue
                if g_u != gB.get(u, float('inf')):
                    stale_pops_b += 1
                    continue
                closedB.add(u)
                backward_expanded += 1
                improve_UB_via_node(u)
                relax_dir(u, g_u, self.rgraph, gB, gF, predB, hB, openB, False)

            # Termination: when best possible connection cannot beat current UB
            if (top_key(openF) + top_key(openB)) >= UB:
                break

        if not meet or not math.isfinite(UB):
            return None

        # Reconstruct forward path
        def build_forward_path_to(u):
            seq = [u]
            while predF[seq[-1]] is not None:
                seq.append(predF[seq[-1]])
            seq.reverse()
            return seq
```

```
    def build_forward_path_from_using_B(v):
        # Use predB chain: next forward node after x is predB[x]
        seq = [v]
        cur = v
        while cur in predB and predB[cur] is not None:
            nxt = predB[cur]
            seq.append(nxt)
            cur = nxt
        return seq

    if meet[0] == 'node':
        m = meet[1]
        path = build_forward_path_to(m)
        tail = build_forward_path_from_using_B(m)
        path += tail[1:]  # skip duplicate m
    else:
        u, v = meet[1], meet[2]  # forward edge u->v
        head = build_forward_path_to(u)
        mid_tail = build_forward_path_from_using_B(v)  # starts with v
        path = head + mid_tail  # includes u->v

    pr = self._calc_stats(path, profile, is_night)
    pr.nodes_expanded_forward = forward_expanded
    pr.nodes_expanded_backward = backward_expanded
    pr.edges_scanned_forward = forward_edges_scanned
    pr.edges_scanned_backward = backward_edges_scanned
    pr.stale_pops_forward = stale_pops_f
    pr.stale_pops_backward = stale_pops_b
    return pr
#                       State & JSON export
```

6) Uniform metric computation (comparable times across profiles)

```python
def _calc_stats(self, path: List[str], profile: str, is_night: bool) -> PathResult:
    dist = 0.0; sig = 0; cross = 0; safepen = 0.0; cost = 0.0; tsec = 0.0
    for i in range(len(path)-1):
        u, v = path[i], path[i+1]
        # find matching edge in forward graph
        edge = next((e for e in self.graph[u] if e['to_node'] == v), None)
        if not edge:  # fallback (shouldn't happen)
            continue
        dist += edge['distance']
        # costs
        if profile == 'shortest':
            c = self._cost_shortest(edge, u)
        elif profile == 'few_traffic_lights':
            c = self._cost_few_traffic_lights(edge, u)
        elif profile == 'safest':
            c = self._cost_safest(edge, u, is_night)
        elif profile == 'fastest':
            c = self._cost_fastest(edge, u)
        else:
            c = self._cost_shortest(edge, u)
        cost += c
        safepen += self._cost_safest(edge, u, is_night) - edge['distance']
        tsec += self._cost_fastest(edge, u)

        # node-based counters on v
        nt = self.nodes[v].tags
        if nt.get('highway') == 'traffic_signals' or nt.get('crossing') == 'traffic_signals':
            sig += 1
        elif nt.get('highway') == 'crossing' or ('crossing' in nt):
            cross += 1

    descr = {
        'shortest': 'shortest path',
        'few_traffic_lights': 'few_traffic_lights path',
        'safest': f"safest path",
        'fastest': 'fastest path',
    }[profile]

    return PathResult(
        path=path, total_cost=cost, distance_meters=dist,
        num_traffic_signals=sig, num_crossings=cross,
        safety_penalties=safepen, time_seconds=tsec,
        description=descr
    )
```