```python
    if len(path) < 2:
        return 0
    total = 0
    for i in range(len(path) - 1):
        r1, c1 = path[i]
        r2, c2 = path[i + 1]
        if abs(r1 - r2) + abs(c1 - c2) != 1:  # Adjacent?
            return float('inf')
        total += gw.get_cost(r2, c2)  # Cost to enter next cell
    return total

def generate_random_path(gw, start, goal, max_length=100):
    """
    Generate initial random path for local search.
    """
    path = [start]
    current = start
    while current != goal and len(path) < max_length:
        neighbors_list = list(gw.neighbors(current[0], current[1]))
        if not neighbors_list:
            break
        nr, nc, _ = random.choice(neighbors_list)
        next_pos = (nr, nc)
        path.append(next_pos)
        current = next_pos
    if current == goal:
        return path
    else:
```

[1]:

```python
import heapq
from collections import defaultdict
import time
import math
import random

class GridWorld:
    """
    2D grid world with varying terrain costs and static obstacles.
    """
    def __init__(self, grid, start, goal):
        self.grid = [row[:] for row in grid]  # Deep copy
        self.rows = len(grid)
        self.cols = len(grid[0])
        self.start = start  # Tuple (r, c)
        self.goal = goal
        # Min cost for heuristic
        self.min_cost = min((c for row in grid for c in row if c > 0))

    def is_valid(self, r, c):
        return 0 <= r < self.rows and 0 <= c < self.cols and self.grid[r][c] > 0

    def get_cost(self, r, c):
        return self.grid[r][c]

    def neighbors(self, r, c):
        dirs = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # 4-connected
        for dr, dc in dirs:
            nr, nc = r + dr, c + dc
            if self.is_valid(nr, nc):
```

```python
    return True

def simulated_annealing(gw, start, goal, max_iter=500, initial_temp=50, cooling_rate=0.99):
    """
    Local search: SA for approximate optimization; replans on dynamic changes.
    """
    current_path = generate_random_path(gw, start, goal)
    current_cost = path_cost(gw, current_path)
    best_path = current_path[:]
    best_cost = current_cost
    temp = initial_temp
    nodes_expanded = 0
    start_time = time.time()
    for i in range(max_iter):
        new_path = perturb_path(current_path)
        if is_valid_path(gw, new_path):
            new_cost = path_cost(gw, new_path)
            delta = new_cost - current_cost
            if delta < 0 or random.random() < math.exp(-delta / temp):
                current_path = new_path
                current_cost = new_cost
                if new_cost < best_cost:
                    best_cost = new_cost
                    best_path = new_path[:]
        temp *= cooling_rate
        nodes_expanded += 1
    end_time = time.time()
    exec_time = end_time - start_time
    path_length = len(best_path)
    return best_cost, path_length, nodes_expanded, exec_time, best_path

def create_maps():
```

```python
        if abs(current[0] - goal[0]) + abs(current[1] - goal[1]) == 1 and gw.is_valid(goal[0], goal[1]):
            path.append(goal)
            return path
    return [start]  # Fallback

def perturb_path(path):
    """
    Perturb path by random adjacent swap (hill-climb like).
    """
    if len(path) < 3:
        return path
    i = random.randint(1, len(path) - 2)
    dirs = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    dr, dc = random.choice(dirs)
    r, c = path[i]
    nr, nc = r + dr, c + dc
    if 0 <= nr < 20 and 0 <= nc < 20:  # Bound check (extend for larger grids)
        new_path = path[:i] + [(nr, nc)] + path[i + 1:]
        return new_path
    return path

def is_valid_path(gw, path):
    """
    Validate path: starts/ends correctly, adjacent, no obstacles.
    """
    if path[0] != gw.start or path[-1] != gw.goal:
        return False
    for i in range(len(path) - 1):
        r1, c1 = path[i]
        r2, c2 = path[i + 1]
        if not gw.is_valid(r1, c1) or not gw.is_valid(r2, c2):
            return False
```

```python
                path.append(current)
                current = came_from[current]
            path.reverse()
            total_cost = g_score[goal]
            exec_time = end_time - start_time
            return total_cost, len(path), nodes_expanded, exec_time, path
        for nr, nc, edge_cost in gw.neighbors(r, c):
            neighbor = (nr, nc)
            tent_g = g_score[current] + edge_cost
            if tent_g < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tent_g
                heapq.heappush(frontier, (tent_g, nr, nc))
    return None

def a_star_search(gw):
    """
    Informed: A* with admissible heuristic.
    """
    start = gw.start
    goal = gw.goal
    h_func = lambda r, c: manhattan_heuristic(r, c, goal, gw.min_cost)
    frontier = []
    heapq.heappush(frontier, (h_func(start[0], start[1]), 0, start[0], start[1]))  # f, g, r, c
    came_from = {}
    came_from[start] = None
    g_score = defaultdict(lambda: float('inf'))
    g_score[start] = 0
    f_score = defaultdict(lambda: float('inf'))
    f_score[start] = h_func(start[0], start[1])
    nodes_expanded = 0
    start_time = time.time()
    visited = set()
```

```python
                yield nr, nc, self.get_cost(nr, nc)  # Position and edge cost to enter

def manhattan_heuristic(r, c, goal, min_cost):
    gr, gc = goal
    return (abs(r - gr) + abs(c - gc)) * min_cost  # Admissible

def uniform_cost_search(gw):
    """
    Uninformed: UCS for optimal path under varying costs.
    """
    start = gw.start
    goal = gw.goal
    frontier = []
    heapq.heappush(frontier, (0, start[0], start[1]))  # g, r, c
    came_from = {}
    g_score = defaultdict(lambda: float('inf'))
    g_score[start] = 0
    came_from[start] = None
    nodes_expanded = 0
    start_time = time.time()
    visited = set()
    while frontier:
        current_cost, r, c = heapq.heappop(frontier)
        current = (r, c)
        if current in visited:
            continue
        visited.add(current)
        nodes_expanded += 1
        if current == goal:
            end_time = time.time()
            # Reconstruct path
            path = []
            while current is not None:
```

```
--- SMALL Map ---
Size: 5x5
UCS: cost=8, path_len=9, nodes=16, time=0.0000
A*: cost=8, path_len=9, nodes=16, time=0.0000
SA: cost=0, path_len=1, nodes=500, time=0.0000

--- MEDIUM Map ---
Size: 10x10
UCS: cost=18, path_len=19, nodes=92, time=0.0010
A*: cost=18, path_len=19, nodes=92, time=0.0000
SA: cost=0, path_len=1, nodes=500, time=0.0000

--- LARGE Map ---
Size: 20x20
UCS: cost=38, path_len=39, nodes=311, time=0.0020
A*: cost=38, path_len=39, nodes=279, time=0.0020
SA: cost=0, path_len=1, nodes=500, time=0.0010

--- DYNAMIC Map ---
Size: 5x5
UCS: cost=8, path_len=9, nodes=16, time=0.0000
A*: cost=8, path_len=9, nodes=16, time=0.0000
SA: cost=16, path_len=17, nodes=500, time=0.0105

--- Dynamic Replanning Demo ---
Initial path using A*: [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4), (2, 4), (3, 4), (4, 4)]
After 3 steps, agent at: (0, 3)
Obstacle appears (e.g., moving vehicle blocks (0,4)); replan with SA:
Replanned path (positions): [(0, 3), (0, 2), (0, 3), (0, 2), (0, 3), (0, 2), (0, 3), (0, 2), (0, 3), (0, 2), (0, 3), (0, 2), (0, 1), (0, 2), (0, 3), (0,
2), (0, 1), (0, 0), (0, 1), (0, 2), (0, 3), (0, 2), (0, 1), (0, 2), (0, 3), (0, 2), (0, 3), (0, 2), (0, 1), (0, 0), (0, 1), (0, 2), (0, 1), (0, 0), (1,
0), (0, 0), (1, 0), (2, 0), (1, 0), (0, 0), (0, 1), (0, 0), (0, 1), (0, 0), (1, 0), (0, 0), (1, 0), (2, 0), (1, 0), (2, 0), (3, 0), (2, 0), (3, 0), (4,
0), (4, 1), (4, 0), (3, 0), (4, 0), (4, 1), (4, 0), (4, 1), (4, 2), (4, 3), (4, 2), (4, 3), (4, 4)]
Replan cost: 65, nodes: 500
```

```
    while frontier:
        _, g, r, c = heapq.heappop(frontier)
        current = (r, c)
        if current in visited:
            continue
        visited.add(current)
        nodes_expanded += 1
        if current == goal:
            end_time = time.time()
            path = []
            while current is not None:
                path.append(current)
                current = came_from[current]
            path.reverse()
            total_cost = g_score[goal]
            exec_time = end_time - start_time
            return total_cost, len(path), nodes_expanded, exec_time, path
        for nr, nc, edge_cost in gw.neighbors(r, c):
            neighbor = (nr, nc)
            tent_g = g_score[current] + edge_cost
            if tent_g < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tent_g
                f = tent_g + h_func(nr, nc)
                f_score[neighbor] = f
                heapq.heappush(frontier, (f, tent_g, nr, nc))
    return None

def path_cost(gw, path):
    """
    Compute total cost of a path.
    """
```