E-fólio B | Folha de resolução para E-fólio

UNIDADE CURRICULAR: Introdução à Inteligência Artificial

CÓDIGO: 21071

DOCENTE: José Coelho

A preencher pelo estudante

NOME: Francisco Silveiro Cardoso

N.º DE ESTUDANTE: 2303219

CURSO: Licenciatura Engenharia Informática

DATA DE ENTREGA: 16/05/2025

Critérios	Auto-avaliação:
Análise (1 valor)	1
Algoritmos (1 valor)	0,8
Resultados (2 valores)	1,7

Auto-avaliação de acordo com o avaliador: +0.1 na nota do e-fólio

Critérios de correção no enunciado.

TRABALHO / RESOLUÇÃO:

Análise do problema

O problema proposto envolve desafios complexos de otimização e gestão de restrições temporais e espaciais. O objetivo duplo, orientar visitantes e maximizar a satisfação do passeio, exige uma abordagem que integre planeamento de caminhos, gestão de recursos temporais, e modelagem de preferências humanas.

O sistema deve equilibrar três componentes essenciais:

- Exploração: Recompensa por visitar células inéditas.
- Visitas eficientes de pontos de interesse: Priorização de células com valor alto.
- Penalização por visitas repetitivas (revisitas consecutivas).

Esses objetivos são conflituosos entre si: explorar áreas novas pode afastar o guia de pontos de interesse, enquanto focar apenas em pontos de interesse pode levar a percursos monótonos. A solução precisa de um modelo matemático que quantifique trade-offs entre esses fatores.

O tempo total do passeio é fixo, e cada movimento consome tempo:

- Terrenos irregulares (células :) dobram o custo temporal.
- A saída final deve ser garantida dentro do limite, mesmo que reste pouco tempo.
- A distância dinâmica até a saída mais próxima precisa ser considerada em tempo real.

Isso exige não só um cálculo prévio de rotas de fuga (para orientação), mas também uma integração contínua dessa informação na decisão do próximo movimento durante o passeio.

O parque é modelado como um grid com:

- Portas: A localização central das portas em cada lado cria padrões de movimento não triviais.

- Obstáculos estáticos: Células inacessíveis (#) fragmentam o espaço, exigindo algoritmos de pathfinding.

- Variação de terreno: Células normais (1 minuto) vs. irregulares (2 minutos) aumentam a complexidade do cálculo de rotas.

A satisfação não é puramente racional:

- Novidade gera satisfação, mas requer exploração arriscada (potencialmente afastando-se de saídas).

- Revisitas consecutivas causam desgaste progressivo.

- Pontos de interesse têm valor fixo, mas a sua localização pode exigir desvios custosos.

Isto implica a necessidade de uma função de avaliação não linear, onde a utilidade de cada ação depende do histórico completo do passeio.

Para parques grandes (ex: 15×15 células):

- O espaço de estados cresce exponencialmente.

- Técnicas como força bruta ou programação dinâmica tornam-se inviáveis.

- É essencial usar heurísticas admissíveis e estratégias de poda inteligentes para reduzir o espaço de busca.

- Concentração de pontos de interesse: Aglomerações podem criar "armadilhas" locais, onde o algoritmo fica preso otimizando localmente.
- Caminhos sem saída: Sequências de células que obrigam a revisitas, mesmo com tempo restante.

O problema combina elementos de pathfinding, otimização sob restrições, e teoria de decisão, com a adição de modelagem comportamental. A solução ideal requer um algoritmo de busca informada (como o A* que utilizei) com heurísticas personalizadas, pré-processamento de dados espaciais para acelerar decisões, mecanismos para evitar otimizações locais e armadilhas de revisitas e paralelização para lidar com a explosão combinatória.

A implementação fornecida aborda muitos desses pontos, mas em cenários do mundo real (como parques gigantes ou restrições dinâmicas), técnicas adicionais como aprendizado por reforço ou decomposição espacial seriam necessárias para manter ou aumentar a eficiência.

Identificação do algoritmo implementado

O algoritmo desenvolvido para otimizar o passeio no parque baseia-se numa versão paralelizada e adaptada do A*. A essência da abordagem reside numa combinação inteligente de exploração estratégica e cálculo preditivo, sustentada por uma heurística multifatorial que avalia simultaneamente múltiplas dimensões do problema.

A arquitetura paralela do sistema permite que quatro processos independentes explorem simultaneamente o parque a partir das diferentes portas de acesso, cada um mantendo seu próprio contexto de busca. Essa estratégia aproveita a natureza distribuída do problema, onde soluções ótimas podem emergir de diferentes pontos de partida. A coordenação entre threads é feita através de variáveis atómicas que atualizam continuamente o melhor caminho global encontrado, garantindo eficiência na utilização de recursos computacionais.

Cada estado da busca é representado de forma compacta através de uma combinação de coordenadas espaciais, máscaras de bits para pontos de interesse visitados e estruturas especializadas para traçar o histórico de visitas às células. Dois bitsets otimizados registam se uma célula foi visitada uma ou múltiplas vezes, enquanto um contador monitora sequências de revisitas consecutivas. Essa representação minimiza o consumo de memória e permite operações rápidas de comparação e atualização.

Configurações-chave como o peso inicial da heurística e uma taxa de decaimento exponencial foram calibradas para equilibrar a exploração inicial com a explotação gradual de rotas promissoras. Limites práticos de 10 segundos por thread e 1 milhão de expansões foram impostos pelo enunciado.

A penalização por revisitas consecutivas (-1 ponto a partir da segunda repetição) e o bónus por novas descobertas (+1 ponto) modelam diretamente as observações comportamentais descritas no problema. Já o tratamento especial para pontos de interesse

próximos ao tempo limite reflete uma estratégia consciente de avaliação de risco, priorizando recompensas imediatas quando o fim do passeio se aproxima.

Resultados

Melhores soluções

Instância	Algoritmo	Configuração	Resultado	Tempo (msec)
1	A*	1	6	18
2	A*	1	8	1438
3	A*	1	9	480
4	A*	1	7	3393
5	A*	1	15	3251
6	A*	1	32	10001
7	A*	1	86	10001
8	A*	1	43	10000
9	A*	1	32	10001
10	A*	1	70	10003

Instância 1

```
==== Instância 1 (N=5 W=4 K=6 T=10) ====
Melhor sat: 10  custo: 6  exp: 408  cpu: 0.018s
Caminho:
(0,2) (1,2) (0,2) (0,1) (0,0) (1,0) (2,0) (2,1) (2,0)
```

Instância 2

```
==== Instância 2 (N=5 W=4 K=5 T=20) ====
Melhor sat: 17  custo: 8  exp: 30328  cpu: 1.438s
Caminho:
(0,2) (0,3) (0,4) (0,3) (1,3) (1,2) (1,1) (0,1) (0,0) (1,0) (2,0) (2,1) (3,1) (3,2) (4,2)
```

Instância 3

```
==== Instância 3 (N=7 W=5 K=10 T=15) ====
Melhor sat: 16  custo: 9  exp: 12624  cpu: 0.480s
Caminho:
(0,3) (0,2) (0,1) (1,1) (1,0) (2,0) (3,0) (4,0) (4,1) (5,1) (5,2) (5,3) (6,3)
```

Instância 4

```
==== Instância 4 (N=7 W=5 K=10 T=20) ====
Melhor sat: 23  custo: 7  exp: 76336  cpu: 3.393s
Caminho:
(3,6) (2,6) (1,6) (0,6) (0,5) (1,5) (1,4) (1,3) (1,2) (1,1) (0,1) (0,0) (1,0) (2,0) (3,0) (3,1) (3,2) (3,1) (3,0)
```

Instância 5

```
==== Instância 5 (N=9 W=6 K=19 T=19) ====
Melhor sat: 23  custo: 15  exp: 102902  cpu: 3.251s
Caminho:
(0,4) (0,5) (0,6) (0,7) (0,8) (1,8) (2,8) (3,8) (4,8) (5,8) (6,8) (6,7) (7,7) (8,7) (8,6) (8,5) (8,4)
```

Instância 6

```
==== Instância 6 (N=11 W=8 K=19 T=48) ====
Melhor sat: 35  custo: 32  exp: 270551  cpu: 10.001s
Caminho:
(10,5) (9,5) (8,5) (7,5) (6,5) (7,5) (8,5) (8,6) (7,6) (7,7) (6,7) (6,8) (5,8) (6,8) (5,8) (4,8) (4,7) (4,8) (4,7)

(5,7) (5,6) (4,6) (3,6) (3,5) (2,5) (1,5) (0,5) (0,6) (0,7) (1,7) (1,8) (2,8) (2,9) (3,9) (3,10) (4,10) (5,10)
```

Instância 7

```
==== Instância 7 (N=13 W=10 K=10 T=120) ====
Melhor sat: 44  custo: 86  exp: 260017  cpu: 10.001s
Caminho:
(12,6) (12,5) (12,4) (12,3) (12,2) (11,2) (12,2) (12,3) (12,4) (12,5) (12,6) (12,7) (11,7) (10,7) (9,7) (8,7) (7,7) (7,6)

(6,6) (6,5) (5,5) (5,4) (5,5) (5,4) (4,4) (4,3) (4,2) (5,2) (6,2) (6,3) (7,3) (7,4) (8,4) (8,5) (9,5) (8,5) (8,4) (7,4) (7,3) (6,3) (6,2) (5,2)

(6,2) (6,3) (7,3) (7,4) (8,4) (8,5) (9,5) (8,5) (8,4) (7,4) (7,3) (6,3) (6,2) (5,2) (4,2) (4,3) (4,4) (3,4) (2,4)

(1,4) (1,5) (1,4) (1,5) (2,5) (2,4) (2,5) (3,5) (3,4) (3,5) (3,6) (4,6) (3,6) (4,6) (4,7) (5,7) (5,8) (5,9) (6,9) (6,10) (6,9) (5,9) (5,8) (5,7)

(3,7) (2,7) (1,7) (0,7) (1,7) (0,7) (0,8) (0,9) (0,10) (0,11) (0,12) (0,11) (0,10) (0,9) (1,9) (2,9) (3,9) (3,10) (4,10) (4,11) (4,12)

(5,12) (6,12) (7,12) (8,12) (9,12) (9,11) (10,11) (9,11) (8,11) (8,12) (7,12) (6,12)
```

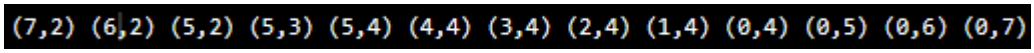
Instância 8

```
==== Instância 8 (N=15 W=15 K=38 T=30) ====
Melhor sat: 25  custo: 43  exp: 272737  cpu: 10.000s
Caminho:
(7,0) (6,0) (5,0) (4,0) (3,0) (2,0) (1,0) (0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0) (9,0) (10,0) (11,0) (12,0) (13,0) (14,0) (14,1)

(14,2) (14,3) (14,4) (14,5) (14,6) (14,7)
```

Instância 9

```
==== Instância 9 (N=15 W=15 K=38 T=45) ====
Melhor sat: 51  custo: 32  exp: 261257  cpu: 10.001s
Caminho:
(7,14) (8,14) (9,14) (10,14) (11,14) (11,13) (11,12) (11,11) (11,12) (11,11) (11,10) (11,9) (11,8) (10,8) (9,8) (8,8) (8,7) (8,6)

(9,6) (9,5) (9,4) (9,3) (9,2) (8,2) (7,2) (6,2) (5,2) (5,3) (5,4) (4,4) (3,4) (2,4) (1,4) (0,4) (0,5) (0,6) (0,7)
```

Instância 10



Anexos

```
#include <bits/stdc++.h>  // Inclui todas as bibliotecas padrão
#include <atomic>         // Para operações atomicas
#include <mutex>          // Para exclusão mútua entre threads
#include <cstdint>        // Tipos inteiros de tamanho fixo


using namespace std;
using pii = pair<int, int>;
static const int INF = 1e9; // Valor infinito para representar obstáculos
const double INIT_WEIGHT = 1.5; // Peso inicial da heurística
const double WEIGHT_DECAY = 0.998; // Decaimento do peso ao longo do tempo


int dr[4] = {-1, 1, 0, 0}, dc[4] = {0, 0, -1, 1}; // Movimentos (cima, baixo, esq, dir)


int N, Tlim, W, Ksum;             // Dimensões, tempo limite, interesses, soma K
vector<string> mapa;              // Mapa do grid (caracteres)
vector<pii> interest;             // Posições dos pontos de interesse
vector<int> interestValue;        // Valores dos pontos de interesse
```

```
vector<vector<int>> distToExit;   // Distância mínima para saídas
vector<vector<int>> interestDist; // Distância mínima para interesses
int total_traversable;            // Total de células transitáveis

// COMPONENTES PARA PARALELIZAÇÃO
atomic<int> globalBestSat(-INF);  // Melhor satisfação global (thread-safe)
atomic<int> globalExpansions(0);  // Contador de expansões de nós
atomic<double> globalCpu(0.0);    // Tempo de CPU consumido
vector<pii> globalBestPath;       // Melhor caminho encontrado
mutex bestMutex;                  // Mutex para acesso ao melhor caminho

inline int moveCost(char ch) { return (ch == '#' ? INF : (ch == ':' ? 2 : 1));}

// Calcula distâncias mínimas para as saídas usando Dijkstra com priority
queue
// Atualiza a matriz distToExit com as distâncias mínimas
void computeExitDistances() {
    distToExit.assign(N, vector<int>(N, INF));
    priority_queue<pair<int, pii>, vector<pair<int, pii>>, greater<>> pq;
    int mid = N / 2;
    vector<pii> doors = {{0, mid}, {N-1, mid}, {mid, 0}, {mid, N-1}};
    for (auto [r, c] : doors) {
```

```
        if (mapa[r][c] == '#') continue;
        distToExit[r][c] = 0;
        pq.emplace(0, make_pair(r, c));
    }
    while (!pq.empty()) {
        auto [d, pos] = pq.top(); pq.pop();
        auto [r, c] = pos;
        if (d > distToExit[r][c]) continue;
        for (int k = 0; k < 4; k++) {
            int nr = r + dr[k], nc = c + dc[k];
            if (nr < 0 || nr >= N || nc < 0 || nc >= N) continue;
            int cost = moveCost(mapa[nr][nc]);
            if (distToExit[r][c] + cost < distToExit[nr][nc]) {
                distToExit[nr][nc] = distToExit[r][c] + cost;
                pq.emplace(distToExit[nr][nc], make_pair(nr, nc));
            }
        }
    }
}
```

```
// Heurística:
    // Valor dos interesses próximos (com bonus para proximidade)
    // Máximo valor restante possível (admissível)
    // Potencial de exploração de células não visitadas
    // Penalidade por revisitas consecutivas
    // Considera o tempo restante e distância para saída mais próxima
    // Prioriza exploração apenas se houver tempo suficiente para sair
int heuristic(int r, int c, int t, int mask, const bitset<225>& visited_once, const bitset<225>& visited_twice, int consec) {
    int timeLeft = Tlim - t;
    int h = 0;
    int max_remaining_value = 0;

    for (int i = 0; i < W; i++) {
        if (!(mask & (1 << i))) {
            auto [ri, ci] = interest[i];
            int dist = interestDist[r][c];
            int exit_time = distToExit[ri][ci];

            if (dist != INF && dist + exit_time <= timeLeft) {
                int value = interestValue[i];
                if (dist <= timeLeft/3) value *= 2;
```

```
            h += value;
            max_remaining_value = max(max_remaining_value, interestValue[i]);
        }
    }
}

h += max_remaining_value;

int min_exit_dist = distToExit[r][c];
int available_exploration_time = timeLeft - min_exit_dist;

int reachable_cells = 0;
if (available_exploration_time > 0) {
    int potential_new_cells = total_traversable -
                        static_cast<int>(visited_once.count()) -
                        static_cast<int>(visited_twice.count());
    reachable_cells = min(available_exploration_time, potential_new_cells);
}
h += reachable_cells;

if (consec >= 2) {
```

```
        h -= 3 * (consec - 1);
    }

    return h;
}

// Estado do nó na busca
struct State {
    int r, c, t, mask, consec; // Posição (r,c), tempo, máscara de interesses coletados
    bitset<225> visited_once;  // Controlo de visitas
    bitset<225> visited_twice; // Controlo de visitas
    int g, f;                  // Valores g (custo real) e f (g + heurística)
    bool operator<(const State& o) const { return f > o.f; } // Operador < para ordenar na priority_queue
};

// Chave única para o estado (usada no hashmap)
struct Key {
    // Componentes essenciais do estado para comparação
    int r, c, t, mask, consec;
    bitset<225> visited_once;
```

```
    bitset<225> visited_twice;
    // Operador == para verificação de igualdade
    bool operator==(const Key& o) const {
        return tie(r, c, t, mask, consec) == tie(o.r, o.c, o.t, o.mask, o.consec)
            && visited_once == o.visited_once
            && visited_twice == o.visited_twice;
    }
};

// Função auxiliar para combinar hashes
static inline void hash_combine(size_t& seed, size_t value) {
    seed ^= value + 0x9e3779b9 + (seed << 6) + (seed >> 2);
}

// Função de hash personalizada para a chave
struct KeyHash {
    // Combina hashes de todos componentes do estado
    // Trata bitsets de forma eficiente usando blocos de 64 bits
    size_t operator()(const Key& k) const {
        size_t h = 0;
```

```
// Hash primitive fields
hash_combine(h, hash<int>{}(k.r));
hash_combine(h, hash<int>{}(k.c));
hash_combine(h, hash<int>{}(k.t));
hash_combine(h, hash<int>{}(k.mask));
hash_combine(h, hash<int>{}(k.consec));

// Fast bitset hashing using 64-bit blocks
auto hash_bitset = [](const bitset<225>& bs) {
    size_t hash = 0;
    for (size_t i = 0; i < 225; i += 64) {
        uint64_t chunk = 0;
        for (size_t j = 0; j < 64 && (i + j) < 225; ++j) {
            chunk |= static_cast<uint64_t>(bs[i + j]) << j;
        }
        hash_combine(hash, std::hash<uint64_t>{}(chunk));
    }
    return hash;
};

hash_combine(h, hash_bitset(k.visited_once));
```

```
        hash_combine(h, hash_bitset(k.visited_twice));

        return h;
    }
};

struct NodeInfo {
    int g;
    Key parent;
    int parentDir;
};

void workerAstar(int doorIdx) {
    // Implementação paralela do A*:
    // Inicializa a partir de uma entrada específica
    // Usa priority_queue para fronteira de busca
    // Mantém hashmap de estados visitados
    // Expande nós considerando movimentos válidos
    // Atualiza melhor caminho encontrado
    // Usa limite 10s ou 1000000 expansões como limite
    auto t0 = chrono::steady_clock::now();
```

```
int expansions = 0;

priority_queue<State> pq;

unordered_map<Key, NodeInfo, KeyHash> info;


int mid = N / 2;

vector<pii> doors = {{0, mid}, {N-1, mid}, {mid, 0}, {mid, N-1}};

auto [r0, c0] = doors[doorIdx];

if (mapa[r0][c0] == '#') return;


int t_init = 1;

int mask0 = 0, g0 = 0;

bitset<225> visited_once0, visited_twice0;

visited_once0.set(r0 * N + c0);


// Initial interest collection

for (int i = 0; i < W; i++) {

    if (interest[i] == make_pair(r0, c0)) {

        mask0 |= (1 << i);

        g0 += interestValue[i];

    }

}
```

```
g0 += 2;  // Initial cell bonus plus exit bonus


// Initialization
double initial_weight = INIT_WEIGHT * pow(WEIGHT_DECAY, t_init);
int h0 = heuristic(r0, c0, t_init, mask0, visited_once0, visited_twice0, 0);
int f0 = g0 + int(initial_weight * h0);


Key k0{r0, c0, t_init, mask0, 0, visited_once0, visited_twice0};
pq.push({r0, c0, t_init, mask0, 0, visited_once0, visited_twice0, g0, f0});
info[k0] = {g0, k0, -1};


int bestSat = -INF;
Key bestKey;


while (!pq.empty()) {
    if (++expansions > 1000000) break;
    auto now = chrono::steady_clock::now();
    if (chrono::duration<double>(now - t0).count() > 10.0) break;


    State u = pq.top(); pq.pop();
    Key uk{u.r, u.c, u.t, u.mask, u.consec, u.visited_once, u.visited_twice};
```

```
// Early exit check with actual movement cost
bool isExit = (u.r == 0 && u.c == mid) || (u.r == N-1 && u.c == mid) ||
              (u.c == 0 && u.r == mid) || (u.c == N-1 && u.r == mid);
if (isExit) {
    int exit_cost = moveCost(mapa[u.r][u.c]);
    int exit_time = u.t + exit_cost;
    if (exit_time <= Tlim && u.g > bestSat) {
        bestSat = u.g;
        bestKey = uk;
    }
}

for (int d = 0; d < 4; d++) {
    int nr = u.r + dr[d], nc = u.c + dc[d];
    if (nr < 0 || nr >= N || nc < 0 || nc >= N) continue;

    char ch = mapa[nr][nc];
    if (ch == '#') continue;

    int mc = moveCost(ch), nt = u.t + mc;
```

```
if (nt > Tlim) continue;


int cell_idx = nr * N + nc;
if (u.visited_twice.test(cell_idx)) continue;


// Update visitation tracking
bitset<225> n_once = u.visited_once;
bitset<225> n_twice = u.visited_twice;
bool is_new = !n_once.test(cell_idx) && !n_twice.test(cell_idx);


if (n_once.test(cell_idx)) {
    n_once.reset(cell_idx);
    n_twice.set(cell_idx);
} else if (!n_twice.test(cell_idx)) {
    n_once.set(cell_idx);
}


// Update interest collection
int nmask = u.mask;
int ng = u.g;
for (int i = 0; i < W; i++) {
```

```
    if (!(nmask & (1 << i)) && interest[i] == make_pair(nr, nc)) {
        nmask |= (1 << i);
        ng += interestValue[i];
    }
}

// Update score components
if (is_new) ng += 1;
int nconsec = is_new ? 0 : u.consec + 1;
ng -= (nconsec >= 2) ? 1 : 0;  // Progressive penalty

Key nk{nr, nc, nt, nmask, nconsec, n_once, n_twice};
auto it = info.find(nk);
if (it == info.end() || ng > it->second.g) {
    int h = heuristic(nr, nc, nt, nmask, n_once, n_twice, nconsec);
    double current_weight = INIT_WEIGHT * pow(WEIGHT_DECAY, u.t);
    int f = ng + int(current_weight * h);

    // Prune unpromising states
    if (f > globalBestSat.load(memory_order_relaxed)) {
        pq.push({nr, nc, nt, nmask, nconsec, n_once, n_twice, ng, f});
```

```
                info[nk] = {ng, uk, d};
            }
        }
    }
}

// Path reconstruction
vector<pii> path;
if (bestSat > -INF) {
    Key cur = bestKey;
    while (true) {
        path.emplace_back(cur.r, cur.c);
        auto &ni = info[cur];
        if (ni.parentDir < 0) break;
        cur = ni.parent;
    }
    reverse(path.begin(), path.end());
}

// Update global best
double cpu = chrono::duration<double>(chrono::steady_clock::now() -
t0).count();
```

```
    if (bestSat > globalBestSat.load(memory_order_relaxed)) {
        lock_guard<mutex> lk(bestMutex);
        if (bestSat > globalBestSat.load(memory_order_relaxed)) {
            globalBestSat.store(bestSat, memory_order_relaxed);
            globalBestPath = move(path);
            globalExpansions.store(expansions, memory_order_relaxed);
            globalCpu.store(cpu, memory_order_relaxed);
        }
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    struct Inst {
        int ID,N,K,W,T;
        vector<vector<int>> M;
    };

    vector<Inst> instancias = {
```

{1,5,6,4,10,{{1,1,1,1,-1},{1,10,1,2,10},{1,-2,10,2,1},{10,10,1,1,1},{-1,1,1,2,-2}}},

{2,5,5,4,20,{{-1,1,2,2,-1},{1,2,1,1,10},{1,1,10,2,2},{10,2,1,10,1},{-1,2,1,10,-2}}},

{3,7,10,5,15,{{1,1,1,1,10,-2,1},{1,-2,10,1,1,10,1},{1,10,1,10,1,1,1},{1,1,-2,10,1,2,1},{2,1,10,1,2,1,10},{2,1,1,1,2,10,-3},{1,-1,10,1,1,1,1}}},

{4,7,10,5,20,{{1,-2,2,2,2,-2,1},{1,1,1,1,1,1,1},{1,10,10,10,10,10,1},{1,1,-2,10,1,2,1},{2,2,2,10,2,10,10},{2,1,2,10,2,10,-3},{1,-1,2,1,1,1,1}}},

{5,9,19,6,19,{{-2,10,1,1,1,1,-2,1,1},{1,2,-2,10,10,1,1,10,2},{1,1,1,1,1,10,10,1,1},{1,10,10,1,10,-2,1,1,1},{1,1,10,1,1,1,2,10,1},{2,1,1,10,1,2,10,1,1},{10,10,2,2,2,2,10,-3,1},{2,1,1,10,1,2,10,1,10},{-8,10,1,1,1,1,1,1,1}}},

{6,11,19,8,48,{{1,-5,1,10,1,1,1,1,1,1,1},{1,1,10,1,2,1,10,1,1,1,1},{1,10,10,1,1,1,1,10,1,-3,1},{1,2,-1,10,2,2,1,2,10,1,1},{2,1,1,1,10,2,1,1,2,10,1},{1,2,1,1,1,10,-2,1,2,10,1},{2,1,2,10,1,1,10,1,1,10,1},{1,-1,1,10,1,2,1,1,10,2,1},{1,10,10,10,1,2,-1,10,2,2,1},{1,1,1,1,1,2,10,1,1,1,-4},{1,1,-2,2,1,1,1,2,2,2,1}}},

{7,13,10,10,120,{{1,1,1,1,1,10,1,2,1,1,1,1,1},{-1,10,1,10,1,2,10,1,10,2,10,10,1},{10,1,1,10,1,2,10,2,10,1,10,-1,1},{1,1,10,1,1,1,-1,1,10,-1,1,10,10},{1,10,1,1,1,10,1,1,10,10,1,1,1},{1,10,-1,10,1,1,10,1,1,1,10,10,1},{1,10,1,1,10,1,1,10,10,1,1,10,1},{10,10,10,-1,1,10,1,1,10,1,1,10,1},{1,2,1,10,2,1,10,1,10,-1,10,1,1},{1,10,2,1,10,2,10,1,10,1,10,1,1},{1,-1,10,1,2,1,10,1,10,1,1,1,10},{1,10,1,10,10,10,10,1,10,1,10,10,-1},{1,1,1,-1,1,1,1,1,10,1,1,1,1}}},

{8,15,38,15,30,{{-3,1,1,1,1,1,1,1,2,2,2,1,-3,1,-2},{1,1,1,1,1,10,10,10,10,10,1,1,2,1,2},{1,1,1,1,1,2,1,1,1,1,1,1,1,1,-2},{1,1,1,1,1,1,-2,2,1,-3,1,1,1,1,1},{1,1,1,1,1,1,1,2,1,2,1,1,1,1,1},{1,10,1,1,1,2,2,1,2,1,1,1,-1,10,1},{1,10,-3,1,2,1,1,2,1,1,1,1,1,10,1},{1,10,1,1,-

```
2,2,1,1,1,1,2,2,1,10,1},{1,10,1,1,1,1,2,1,1,1,1,1,1,10,2},{1,10,-
2,2,1,2,1,2,1,1,1,1,1,10,1},{1,1,1,1,1,1,1,1,1,1,2,1,1,1,2},{-
2,1,1,2,1,2,1,1,1,2,1,2,1,-
3,1},{1,1,1,1,2,1,1,1,1,1,1,1,1,1,1},{1,2,1,1,1,10,10,10,10,10,1,1,2,1,1},{1,1,-
3,1,1,1,1,1,1,1,1,-3,1,1,-4}}},

    {9,15,38,15,45,{{-3,10,1,1,1,1,1,1,2,2,2,10,-6,1,-
4},{1,10,1,10,1,10,10,10,10,10,1,1,2,10,2},{1,10,1,10,1,2,1,10,1,10,1,10,1,10,-
4},{1,1,1,10,1,10,-4,10,1,-
6,1,10,1,1,1},{1,10,10,10,1,10,1,10,1,10,1,10,10,10,1},{1,10,1,1,1,10,2,10,2,10,
1,10,-2,10,1},{1,10,-6,10,2,10,1,1,1,10,1,10,1,10,1},{1,10,1,10,-
4,10,10,10,10,10,2,2,1,10,1},{1,10,1,10,10,10,2,1,1,10,10,10,1,10,2},{1,10,-
4,2,1,1,1,10,1,10,1,1,1,10,1},{1,10,10,10,1,10,1,10,1,10,2,10,10,10,2},{-
4,1,1,2,1,10,1,10,1,2,1,2,1,-
6,1},{10,10,1,10,2,10,1,10,1,10,1,10,10,10,10},{1,10,1,10,1,10,10,10,10,10,1,10
,2,10,1},{1,1,-6,10,1,1,1,1,1,1,1,-6,1,1,-8}}},

    {10,15,76,15,45,{{-6,10,1,1,1,1,1,1,2,2,2,10,-6,1,-
4},{1,10,1,10,1,10,10,10,10,10,1,1,2,10,2},{1,10,1,10,1,2,1,10,1,10,1,10,1,10,-
4},{1,1,1,10,1,10,-4,10,1,-
6,1,10,1,1,1},{1,10,10,10,1,10,1,10,1,10,1,10,10,10,1},{1,10,1,1,1,10,2,10,2,10,
1,10,-2,10,1},{1,10,-6,10,2,10,1,1,1,10,1,10,1,10,1},{1,10,1,10,-
4,10,10,10,10,10,2,2,1,10,1},{1,10,1,10,10,10,2,1,1,10,10,10,1,10,2},{1,10,-
4,2,1,1,1,10,1,10,1,1,1,10,1},{1,10,10,10,1,10,1,10,1,10,2,10,10,10,2},{-
4,1,1,2,1,10,1,10,1,2,1,2,1,-
6,1},{10,10,1,10,2,10,1,10,1,10,1,10,10,10,10},{1,10,1,10,1,10,10,10,10,10,1,10
,2,10,1},{1,1,-6,10,1,1,1,1,1,1,1,-6,1,1,-8}}}

};


// Precompute interest distances (BFS from all interests)

interestDist.assign(N, vector<int>(N, INF));

queue<pair<int, pii>> q;
```

```
for (auto [r,c] : interest) {
    interestDist[r][c] = 0;
    q.emplace(0, make_pair(r, c));
}

while (!q.empty()) {
    auto [d, pos] = q.front(); q.pop();
    auto [r,c] = pos;
    if (d > interestDist[r][c]) continue;

    for (int k = 0; k < 4; k++) {
        int nr = r + dr[k], nc = c + dc[k];
        if (nr >= 0 && nr < N && nc >= 0 && nc < N && mapa[nr][nc] != '#') {
            if (interestDist[nr][nc] > d + 1) {
                interestDist[nr][nc] = d + 1;
                q.emplace(d + 1, make_pair(nr, nc));
            }
        }
    }
}
```

```
for (auto &ins : instancias) {
    // Processa múltiplas instâncias:
// Carrega dados de cada instância
// Pré-processa distâncias para saídas e interesses
// Inicia threads para cada porta de entrada
// Recolhe resultados e exibe estatísticas
// Executa para as 10 instâncias pré-definidas

// Estrutura Inst armazena configurações de cada problema
// Pré-computações são refeitas para cada instância
// Threads são sincronizadas e resultados comparados
    N = ins.N;
    Tlim = ins.T;
    Ksum = ins.K;
    mapa.resize(N);
    interest.clear();
    interestValue.clear();

    // Parse map and interests
    for (int r = 0; r < N; r++) {
        string s;
```

```
    for (int c = 0; c < N; c++) {
        int v = ins.M[r][c];
        if (v == 10) s.push_back('#');
        else if (v == 2) s.push_back(':');
        else if (v == 1) s.push_back('.');
        else {
            s.push_back('0' + (-v));
            interest.emplace_back(r, c);
            interestValue.push_back(-v);
        }
    }
    mapa[r] = s;
}
W = interest.size();

// Calcular traversable cells
total_traversable = 0;
for (int r = 0; r < N; r++)
    for (int c = 0; c < N; c++)
        if (mapa[r][c] != '#') total_traversable++;
```

```
computeExitDistances();


// Precalcular interest distances
interestDist.assign(N, vector<int>(N, INF));
queue<pair<int, pii>> q;
for (auto [r,c] : interest) {
    interestDist[r][c] = 0;
    q.emplace(0, make_pair(r, c));
}


while (!q.empty()) {
    auto [d, pos] = q.front(); q.pop();
    auto [r,c] = pos;
    if (d > interestDist[r][c]) continue;


    for (int k = 0; k < 4; k++) {
        int nr = r + dr[k], nc = c + dc[k];
        if (nr >= 0 && nr < N && nc >= 0 && nc < N && mapa[nr][nc] != '#') {
            if (interestDist[nr][nc] > d + 1) {
                interestDist[nr][nc] = d + 1;
                q.emplace(d + 1, make_pair(nr, nc));
```

```
            }
        }
    }
}

{
    lock_guard<mutex> lk(bestMutex);
    globalBestSat = -INF;
    globalBestPath.clear();
    globalExpansions = 0;
    globalCpu = 0.0;
}

// Lança as threads e resultados
vector<thread> threads;
for (int di = 0; di < 4; di++)
    threads.emplace_back(workerAstar, di);

for (auto &th : threads)
    th.join();
```

```
        int ideal = Tlim + Ksum;
        int cost = ideal - globalBestSat;
        cout << "\n==== Instância " << ins.ID
             << " (N=" << N << " W=" << W << " K=" << Ksum << " T=" << Tlim << ")
====\n";
        cout << "Melhor sat: " << globalBestSat
             << "  custo: " << cost
             << "  exp: " << globalExpansions
             << "  cpu: " << fixed << setprecision(3) << globalCpu << "s\n";
        cout << "Caminho:\n";
        for (auto &p : globalBestPath)
            cout << "(" << p.first << "," << p.second << ") ";
        cout << "\n";
    }
    return 0;
}
```