COMP3702 Artificial Intelligence (Semester 2, 2025)
Assignment 1: Search in CHEESEHUNTER

Name: XIAOYU ZHAO

Student ID: 48064433

Student email: s4806443@uq.edu.au

Note: Please edit the name, student ID number and student email to reflect your identity and **do not modify the design or the layout in the assignment template**, including changing the paging.

Question 1 (Complete your full answer to Question 1 on the remainder page 1)

Dimension	Value	Justification
Planning horizon	Finite, goal-directed, sequential	The level has a clear endpoint (activate all levers and reach the cheese), with the path involving multi-step sequential decision-making; the problem concludes upon reaching the goal, falling under finite horizon sequential planning.
Representation	Discrete, factored state (grid position + trap/lever status)	The state consists of discrete grid coordinates (row, col) and binary trap/lever switch states; this is a typical “factorized” discrete state space, facilitating search and hashing.
Computational limits	Time/space bounded; exponential in worst case	The branching factor increases with actions and feasible successors, leading to exponential node growth in worst-case scenarios. UCS/A*, priority queues, deduplication/optimal g-pruning, and heuristics are required to solve within time and memory constraints.
Learning	No learning required (planning only)	The assignment employs a given environmental model for a one-time solution; it does not require estimating transition/cost parameters from empirical data, constituting a model-based planning problem rather than a learning problem.
Sensing uncertainty	Fully observable	The map, player positions, and mechanism states are fully visible to the searcher; the target verification is_solved(state) is deterministic, with no perceptual noise or partial observability.
Effect uncertainty	Deterministic transitions	Given a valid condition, the transition of an action is deterministic (`perform_action` either fails or produces a unique successor), with no random outcomes or action noise.
Number of agents	Single agent	Only the mouse (agent) interacts with the static environment; there is no strategic coupling with opponents or teammates.
Interactivity	Static environment, offline planning (replanning not required)	The environment does not change autonomously (except for lever/trap state transitions triggered by agents); a single offline planning session can determine the solution with the minimum cost, eliminating the need for online replanning.

Question 2 (Complete your full answer to Question 2 on page 2)

1. Action Space

Definition: Action Space refers to the set of all actions an agent can potentially take in any given state. It defines which operations can be performed from a particular state to transition to the next state.

In CheeseHunter:

- All actions are defined in GameEnv.ACTIONS and GameEnv.ACTION_COST.
- Actions include walk (wl, wr), sprint (sl, sr), jump (j), climb (c), descend (d), and activate mechanism (a).
- Validity is determined via `game_env.perform_action(state, action)`. If it returns `(True, next_state)`, the action is valid.

2. State Space

Definition: State Space is the set of all possible states in a problem. Each state must contain all information that affects future action selection and outcomes.

In CheeseHunter:

- State is represented by the GameState class (in game_state.py).
- Primarily contains:
- Mouse position (row, col)
- Status of each trap (open/closed), stored in trap_status
- This information determines which actions are feasible (e.g., whether a trap can be passed through), thus constituting a complete state description.

3. Transition Function

Definition: The Transition Function defines how the current state changes to the next state after an action is performed.

In CheeseHunter:

- Implemented by `game_env.perform_action(state, action)`.
- Input: the current GameState and an action. Output: (success, next_state).

4. Utility / Cost Function

Definition: The cost function (or the negative of the utility function) measures the path cost from the initial state to the target state. The goal of the search algorithm is to find the path with the lowest total cost.

In CheeseHunter:

- Each action has a fixed cost defined in GameEnv.ACTION_COST.
- For example: Walk = 1.0, Sprint = 1.9, Jump/Climb = 2.0, Fall = 0.5, Activate = 1.0.
- The search algorithm (UCS / A*) accumulates these action costs to determine the total path cost.
- The objective function is to find the path with the lowest total cost, enabling the mouse to activate all mechanisms + reach the cheese.

Question 3 (Complete your full answer to Question 3 on page 3)

Testcase	**Algorithm**	**Path Cost**	**Runtime (s)**	**Expanded (nodes)**	**Generated (nodes)**	**Max Frontier**
1	**UCS**	**17.4**	**0.0001852036**	**17**	**1**	**1**
1	**A***	**17.4**	**0.0001276207**	**17**	**16**	**4**
2	**UCS**	**40.9**	**0.0003731585**	**70**	**1**	**1**
2	**A***	**40.9**	**0.0003903437**	**57**	**61**	**6**
3	**UCS**	**89.0**	**0.0032761574**	**586**	**1**	**1**
3	**A***	**89.0**	**0.0043807507**	**563**	**583**	**33**
4	**UCS**	**115.1**	**0.0222137451**	**4564**	**1**	**1**
4	**A***	**115.1**	**0.0306509972**	**3808**	**3888**	**127**
5	**UCS**	**256.3**	**4.5285696983**	**808292**	**1**	**1**
5	**A***	**256.3**	**11.8668496609**	**743318**	**758779**	**15481**
6	**UCS**	**507.6**	**32.8296630383**	**4172776**	**1**	**1**
6	**A***	**507.6**	**78.7746360302**	**3801933**	**3866860**	**65157**

Summary:

Both UCS and A* always return the same optimal path cost, as expected. On smaller levels (1–4), A* expands fewer nodes and sometimes achieves slightly faster runtime. However, on larger levels (5–6), A* still expands fewer nodes but becomes slower than UCS, due to higher heuristic computation overhead and larger frontier sizes.

Case Analysis:

- Case 1–2: A* reduces expanded nodes marginally. Runtime is slightly better, but frontier size increases.
- Case 3–4: A* expands ~10–20% fewer nodes. Runtime is roughly equal to UCS, but frontier size increases significantly.
- Case 5–6: A* expands ~8–9% fewer nodes, yet runtime is 2–2.5× slower. Frontier size grows sharply (over 15k and 65k), creating memory pressure.

Interpretation:

- The heuristic (action-weighted Manhattan distance) is admissible and ensures optimality.
- However, it is weak in guiding search because it ignores obstacles and lever constraints.
- As a result, A* spends more time computing heuristics than it saves by pruning nodes, especially on large maps.
- UCS, despite exploring more nodes, benefits from simpler per-node computations, making it faster in large cases.

Conclusion:

- A* achieved its theoretical advantage (fewer expanded nodes), but not practical speedup in large-scale problems.
- The trade-off lies in heuristic strength vs computational overhead: weak heuristics are cheap but unhelpful, while slightly more informed heuristics could improve runtime if carefully designed.
- Future improvements may include preprocessing lever positions, memoizing pairwise distances, or introducing admissible lever-aware bounds to strengthen the heuristic without excessive overhead.

Question 4 (Complete your full answer to Question 4 on pages 4 and 5, and keep page 5 blank if you do not need it)

Heuristic Design:

Our heuristic is based on an action-weighted Manhattan distance. For each axis, we use the minimum possible action cost per grid unit:

- Horizontal: min(walk, sprint/2) = 0.95 per tile.
- Upward: min(jump, climb) = 2.0 per tile.
- Downward: drop = 0.5 per tile.

This ensures that the heuristic never overestimates the true cost, since it ignores obstacles and only considers the cheapest possible actions along each axis. For states with unactivated levers, earlier versions attempted to add lever-related costs (e.g., activation costs and lever chaining). However, this occasionally caused overestimation, so the final heuristic was simplified to only estimate the distance from the player to the goal using weighted Manhattan distance.

Admissibility:

The heuristic is admissible, because in every case it provides a cost lower than or equal to the true minimal path cost. For example, if the goal lies below the player, we add exactly one drop cost, even if the vertical distance is multiple rows. This is a strict lower bound, since multiple drops may be required in reality. Similarly, upward and horizontal movements are weighted by the minimal per-tile costs, which are always less than or equal to the actual path cost in the game environment.

Effectiveness:

- On smaller maps, the heuristic is informative: it prioritises paths closer to the goal and leads to fewer node expansions compared to UCS.
- On larger maps, the heuristic is too weak: because it ignores obstacles and lever requirements, the estimate is often far below the true cost, so A* explores almost as many nodes as UCS. This explains why runtime improvement is limited, and sometimes A* is slower than UCS due to heuristic computation overhead.
- Frontier size is consistently larger in A*, since nodes are retained for comparison by f-values.

Limitations:

- Obstacle ignorance: the heuristic only measures straight-line distances, ignoring walls and unreachable spaces.
- Lever ignorance: the heuristic does not account for the mandatory requirement to activate all levers, so it cannot effectively guide the search when lever order is important.
- Computational overhead: although the heuristic is simple, it must be evaluated for every generated node. In large maps with millions of nodes, this becomes costly.

Possible Improvements:

- Lever-aware heuristics: safely incorporate lower bounds for lever activation by precomputing the minimal distance from each lever to the goal, then taking the maximum of (player→goal, player→lever→goal). This would remain admissible.
- Precomputation and caching: store results of distance calculations between grid cells, reducing repeated computation in large maps.
- Consistency optimisations: ensuring the heuristic is consistent would allow us to eliminate node re-openings and reduce runtime further.

- Hybrid strategy: in small maps, current heuristic is efficient; in large maps, one might switch to a simpler “goal distance only” heuristic to reduce per-node overhead.

Conclusion：

The heuristic is admissible and simple, ensuring correctness, but too weak to provide large practical gains in runtime. It reduces expanded nodes modestly but increases frontier and runtime in large testcases. Future work should focus on stronger yet still admissible lever-aware heuristics and efficiency improvements through caching or precomputation.

Appendix/References