Student: Pascal Musabyimana

Course: Reinforcement Learning

Assignment: Shortest Path and Search-Based Optimization

Instructions:

- Part 1 (Dijkstra): See PDF explanation with hand calculations and graph.

- Part 2 (A\*): Run Astar\_assignment.py with pygame installed.

Heuristics: Euclidean, Manhattan, Chebyshev, Weighted Euclidean

- Part 3 (CartPole): Run notebook in Jupyter to reproduce plots and results.

All results were obtained using Python 3.12 with gymnasium==1.0.0 and pygame==2.6.0

Part 1 - Dijkstra's Shortest Path Algorithm

* From **A**: **A → E (190) → H (90) → J (300) = 190 + 90 + 300 = 580**
* From **B** (best): B → D (110) → E (90) → H (90) → J (300) = **590**
* From **C** (best): C → F (240) → I (80) → J (280) = **600**

So the shortest travel time to J is **580** from station **A** via **A–E–H–J**.

**Part 2 - Exploring A\* Pathfinding Heuristics & Parameters**

**Setup.** Grid 45×35, diagonals allowed (cost = √2), start (5,5) → goal (44,25). Heuristics: Euclidean, Manhattan, Chebyshev, Weighted Euclidean (w∈{0.5,1.0,1.5,2.0}). Metrics: path length, time, optimality (vs Dijkstra), nodes explored.

**Key findings from my runs**

**Empty grid**

* Fewest explored nodes: **Manhattan = 40** (optimal ✓).
* Euclidean explored 481; Chebyshev 708; Weighted (w=1.5/2.0) also explored 40 and stayed optimal.
* Interpretation: On a straight shot, stronger goal bias (Manhattan or larger weights) shrinks the search while still hitting the optimal.

**Simple barrier (vertical wall with a gap)**

* Absolute fewest explored: **Weighted w=1.5 → 45** (but **not optimal**; length 48.113 > 47.284).
* Best **optimal** heuristic: **Manhattan → 78** explored (optimal ✓).
* Interpretation: Larger weights (Weighted A\*) become *inadmissible*: they explore less but can detour to a longer path when obstacles mislead the heuristic.

**Maze**

* Fewest explored while optimal: **Weighted w=2.0 → 331** (optimal ✓). Manhattan/Euclidean also optimal but explored more.
* Interpretation: In structured mazes, a stronger heuristic helps push through long corridors efficiently while still landing on the optimal route.

**Scattered obstacles (~20%)**

* Fewest explored overall: **Weighted w=2.0 → 53** (but **not optimal**).
* Fewest explored **while optimal**: **Weighted w=1.5 → 55** (optimal ✓), slightly better than Manhattan (58).
* Interpretation: Moderate weighting (≈1.5) gave the best exploration/optimality trade-off here.

**Bottom line (who “wins” where?)**

* **Open / lightly obstructed:** Manhattan or Weighted (1.5–2.0) for minimal exploration; weights ≥2.0 risk suboptimality around barriers.
* **Single-gap barrier:** Stick to **admissible** heuristics (Manhattan/Euclidean) for guaranteed optimality; heavy weights can go wrong.
* **Maze:** Weighted (2.0) was fastest/leanest *and* still optimal in my run.
* **Random clutter:** **Weighted 1.5** was the best *optimal* performer; **w=2.0** explored the least but missed optimal once.

**Questions to answer (with your data)**

1. **Why does Manhattan work better/worse in certain scenarios?**

* Better when movement is grid-aligned and the route is mostly axial; it gives a strong, simple goal bias and (with diagonals allowed) still remains admissible (it never overestimates).
* Worse when diagonals are critical or obstacles require diagonal detours; Manhattan underestimates diagonal progress more coarsely than Euclidean/octile, so it can expand more or be more easily misled.

1. **When prefer Chebyshev over Euclidean?**

* Prefer **Chebyshev** if diagonal moves cost the **same** as orthogonal (unit cost for 8-neighbors).
* In your setup, diagonals cost **√2**, so a tighter diagonal-aware metric would actually be **octile distance**:

hoctile(dx,dy)=max⁡(dx,dy)+(2−1)min⁡(dx,dy)h\_{\text{octile}}(dx,dy)=\max(dx,dy)+( \sqrt{2}-1)\min(dx,dy)hoctile​(dx,dy)=max(dx,dy)+(2​−1)min(dx,dy)

Euclidean and octile are usually tighter (less underestimation) than Chebyshev when diagonal ≠ orthogonal.

1. **Effect of increasing the heuristic weight (Weighted A\*)**

* Increasing **w > 1** reduces exploration and time (greedier search) but **breaks admissibility** → may return **suboptimal paths**.
* Your results show exactly that: **Simple Barrier** and **Scattered Obstacles (w=2.0)** returned longer paths while exploring very few nodes.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

Screenshots from running astarheuristics.py file. Run it with the command : starheuristics.py –bench for the results.

**Part 2 – Task 3 & 4 (Patterns + Analysis)**

Across patterns, strong goal bias (Manhattan or Weighted 1.5–2.0) minimizes exploration on open or corridor-like maps but risks suboptimal paths when obstacles are curved, concave, or require non-obvious doorways. Euclidean (and Weighted 1.0) provided the most reliable optimality with competitive exploration, while Chebyshev remained admissible yet expanded more due to a looser estimate under √2 diagonal costs. In practice: use moderate weights for speed on open grids, and switch to Euclidean (unweighted) for robustness around complex obstacle geometry.

**Part 3 – Search-Based Optimization (CartPole-v1)**

**Part 1 – Random Action Control**

* **Average reward:** ≈ 22 steps
* **Interpretation:** Random actions make the pole fall quickly; this acts as a performance baseline.
* **A graph of a number of episodes

  Description automatically generated**
* When I controlled the cartpole by taking completely random actions, the pole stayed up for about 22 steps on average. The histogram shows most episodes ended very early, which makes sense because there’s no logic keeping the pole balanced. This run basically acts as a baseline for later methods.

**Part 2 – Angle-Based Control**

* **Average reward:** ≈ 42 steps
* **Interpretation:** Using the pole’s angle for control doubles the lifetime. The policy now reacts to the tilt but remains unstable.
* **A graph with blue lines and a blue line

  Description automatically generated**
* Using the pole’s angle to decide the direction already doubled my performance compared to random actions (≈ 42 timesteps on average). It’s still a very simple heuristic, but the policy reacts to the pole’s tilt, which keeps it upright longer. The histogram shows more episodes above 50 steps, meaning it’s somewhat stable but still fails often.

**Part 3 – Random Search**

* **Best weights:** [0.1167, 0.2187, 0.8714, 0.2236]
* **Average reward:** 500 (maximum possible)
* **Key features:** Pole angle (θ) and angular velocity (θ̇) dominate.
* **Interpretation:** Pure random search eventually found a perfect controller by chance; good for wide exploration but inefficient.

**A graph of black and red dots

Description automatically generated**

After running 1000 random weight samples, I found a weight vector that consistently gets the maximum reward of 500, meaning it balances the pole perfectly for the full episode.

The largest weights are on the angle (θ) and angular velocity (θ̇), showing that keeping track of the pole’s orientation is much more important than the cart’s position.  
The 3D scatter plot (search\_scatter.png) shows that successful weights (red) cluster in a narrow region, while poor ones (black) are scattered everywhere else.

**Part 4 – Hill Climbing**

* **Best weights:** [0.1876, -0.0255, 2.3374, 0.6063]
* **Average reward:** 500 (max)
* **Interpretation:** Rapid convergence to the optimal policy. The largest weight on pole angle shows it’s the most critical feature. Histogram confirms almost every episode reached 500 timesteps.
* A graph with numbers and a bar

  Description automatically generated

**Part 5 – Simulated Annealing (Adaptive)**

* **Best weights:** [0.0012, 0.2987, -0.2741, -0.8906]
* **Average reward:** ≈ 9 steps
* **Interpretation:** Failed to stabilize. The algorithm cooled too quickly and noise σ shrank too much (≈ 0.0001), stopping exploration prematurely.
* Simulated annealing perturbs the weight vector and occasionally accepts *worse* solutions to escape local optima. A new candidate with reward RtR\_tRt​ is accepted if Rt≥Rt−1R\_t \ge R\_{t-1}Rt​≥Rt−1​ (improvement) or with probability exp⁡ ⁣((Rt−Rt−1)/T)\exp\!\big((R\_t - R\_{t-1})/T\big)exp((Rt​−Rt−1​)/T) when it’s worse. The temperature TTT is gradually cooled (e.g., Tk=T0⋅αkT\_k = T\_0 \cdot \alpha^kTk​=T0​⋅αk), so exploration is high early and becomes greedy as T→0T \to 0T→0.
* A graph with numbers and lines

  Description automatically generated

**Comparison & Conclusion**

| **Algorithm** | **Avg Reward** | **Performance** | **Notes** |
| --- | --- | --- | --- |
| Random Policy | ~22 | Poor | Baseline, no control |
| Angle Control | ~42 | Better | Simple reactive logic |
| Random Search | 500 | Excellent | Found perfect weights |
| Hill Climbing | 500 | Excellent | Fast convergence |
| Simulated Annealing | ~9 | Failed | Over-cooled, stuck early |

**Summary:**  
Hill Climbing and Random Search both achieved the maximum reward of 500, perfectly balancing the pole.  
Simulated Annealing underperformed due to aggressive cooling and vanishing noise.  
Across methods, the pole angle (θ) and angular velocity (θ̇) are the dominant control variables.

A graph with a line

Description automatically generated

**Discussion (Part 1.6)**

* **Reward vs Computation:**  
  Random Search explores broadly but wastes samples.  
  Hill Climbing efficiently refines good solutions.  
  Simulated Annealing can escape local optima but needs tuned cooling and noise.
* **Feature Expansion:**  
  Adding extra terms (e.g., θ², θ × θ̇, x × θ) would enrich the linear model and help search methods find better policies faster.
* **MountainCar-v0 Insight:**  
  Linear policies struggle because reward appears only at the goal.  
  Search-based methods need **feature expansion** (tile coding, RBFs) or **reward shaping** (e.g., + |velocity| bonus) to learn momentum.

**Final Takeaway**

Random Search and Hill Climbing both reached optimal control; the key is exploiting angle and angular velocity feedback.  
Simulated Annealing needs slower cooling or higher σ to avoid early freezing.  
Feature expansion can further strengthen linear search-based agents, especially in harder environments like MountainCar.