A screenshot of a computer

Description automatically generated

* 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**.

Task 2

Hello from the pygame community. https://www.pygame.org/contribute.html

(/Users/pascal-maker/pascalworkspace/env) pascal-maker@MacBook-Pro-van-Pascal reinforcementlearning % python astarheuristics.py --bench

pygame 2.5.2 (SDL 2.28.3, Python 3.10.15)

Hello from the pygame community. https://www.pygame.org/contribute.html

=== A\* Benchmark (COLS=45, ROWS=35, diag=True) ===

Start=(5, 5), Goal=(44, 25)

Scenario | Heuristic | Length | Time(ms) | Optimal | Explored

------------------------------------------------------------------------------------------

Empty Grid | Euclidean | 47.284 | 7.09 | YES | 481

Empty Grid | Manhattan | 47.284 | 0.55 | YES | 40

Empty Grid | Chebyshev | 47.284 | 9.13 | YES | 708

Empty Grid | WeightedEuclid w=0.5 | 47.284 | 18.99 | YES | 1402

Empty Grid | WeightedEuclid w=1.0 | 47.284 | 6.89 | YES | 481

Empty Grid | WeightedEuclid w=1.5 | 47.284 | 0.57 | YES | 40

Empty Grid | WeightedEuclid w=2.0 | 47.284 | 0.61 | YES | 40

Simple Barrier | Euclidean | 47.284 | 5.21 | YES | 377

Simple Barrier | Manhattan | 47.284 | 1.11 | YES | 78

Simple Barrier | Chebyshev | 47.284 | 6.58 | YES | 550

Simple Barrier | WeightedEuclid w=0.5 | 47.284 | 15.01 | YES | 1151

Simple Barrier | WeightedEuclid w=1.0 | 47.284 | 5.04 | YES | 377

Simple Barrier | WeightedEuclid w=1.5 | 48.113 | 0.62 | NO | 45

Simple Barrier | WeightedEuclid w=2.0 | 48.113 | 0.67 | NO | 50

Maze | Euclidean | 151.000 | 2.75 | YES | 381

Maze | Manhattan | 151.000 | 2.32 | YES | 374

Maze | Chebyshev | 151.000 | 2.42 | YES | 385

Maze | WeightedEuclid w=0.5 | 151.000 | 7.01 | YES | 395

Maze | WeightedEuclid w=1.0 | 151.000 | 2.74 | YES | 381

Maze | WeightedEuclid w=1.5 | 151.000 | 2.36 | YES | 351

Maze | WeightedEuclid w=2.0 | 151.000 | 2.32 | YES | 331

Scattered Obstacles | Euclidean | 49.627 | 4.28 | YES | 385

Scattered Obstacles | Manhattan | 49.627 | 0.62 | YES | 58

Scattered Obstacles | Chebyshev | 49.627 | 5.93 | YES | 574

Scattered Obstacles | WeightedEuclid w=0.5 | 49.627 | 11.43 | YES | 1111

Scattered Obstacles | WeightedEuclid w=1.0 | 49.627 | 4.36 | YES | 385

Scattered Obstacles | WeightedEuclid w=1.5 | 49.627 | 0.60 | YES | 55

Scattered Obstacles | WeightedEuclid w=2.0 | 50.456 | 0.56 | NO | 53

Done.

Task 3

(/Users/pascal-maker/pascalworkspace/env) pascal-maker@MacBook-Pro-van-Pascal reinforcementlearning % python astarheuristics.py --bench

pygame 2.5.2 (SDL 2.28.3, Python 3.10.15)

Hello from the pygame community. https://www.pygame.org/contribute.html

=== A\* Benchmark (COLS=45, ROWS=35, diag=True) ===

Start=(5, 5), Goal=(44, 25)

Scenario | Heuristic | Length | Time(ms) | Optimal | Explrd

------------------------------------------------------------------------------------------

Empty Grid | Euclidean | 47.284 | 6.33 | YES | 481

Empty Grid | Manhattan | 47.284 | 0.48 | YES | 40

Empty Grid | Chebyshev | 47.284 | 8.79 | YES | 708

Empty Grid | Weighted w=0.5 | 47.284 | 17.84 | YES | 1402

Empty Grid | Weighted w=1.0 | 47.284 | 6.24 | YES | 481

Empty Grid | Weighted w=1.5 | 47.284 | 0.60 | YES | 40

Empty Grid | Weighted w=2.0 | 47.284 | 0.48 | YES | 40

Simple Barrier | Euclidean | 47.284 | 4.74 | YES | 377

Simple Barrier | Manhattan | 47.284 | 1.03 | YES | 78

Simple Barrier | Chebyshev | 47.284 | 6.40 | YES | 550

Simple Barrier | Weighted w=0.5 | 47.284 | 14.51 | YES | 1151

Simple Barrier | Weighted w=1.0 | 47.284 | 4.79 | YES | 377

Simple Barrier | Weighted w=1.5 | 48.113 | 0.56 | NO | 45

Simple Barrier | Weighted w=2.0 | 48.113 | 0.62 | NO | 50

Maze DFS | Euclidean | 219.000 | 7.29 | YES | 506

Maze DFS | Manhattan | 219.000 | 3.40 | YES | 506

Maze DFS | Chebyshev | 219.000 | 3.14 | YES | 506

Maze DFS | Weighted w=0.5 | 219.000 | 3.04 | YES | 508

Maze DFS | Weighted w=1.0 | 219.000 | 3.21 | YES | 506

Maze DFS | Weighted w=1.5 | 219.000 | 3.34 | YES | 506

Maze DFS | Weighted w=2.0 | 219.000 | 3.23 | YES | 498

Scattered Obstacles | Euclidean | 49.627 | 4.23 | YES | 385

Scattered Obstacles | Manhattan | 49.627 | 0.67 | YES | 58

Scattered Obstacles | Chebyshev | 49.627 | 5.80 | YES | 574

Scattered Obstacles | Weighted w=0.5 | 49.627 | 10.89 | YES | 1111

Scattered Obstacles | Weighted w=1.0 | 49.627 | 4.30 | YES | 385

Scattered Obstacles | Weighted w=1.5 | 49.627 | 0.64 | YES | 55

Scattered Obstacles | Weighted w=2.0 | 50.456 | 0.50 | NO | 53

Linear Horizontal | Euclidean | 51.385 | 7.37 | YES | 564

Linear Horizontal | Manhattan | 51.385 | 0.51 | YES | 49

Linear Horizontal | Chebyshev | 51.385 | 8.17 | YES | 652

Linear Horizontal | Weighted w=0.5 | 51.385 | 14.70 | YES | 1256

Linear Horizontal | Weighted w=1.0 | 51.385 | 7.81 | YES | 564

Linear Horizontal | Weighted w=1.5 | 51.385 | 0.62 | YES | 52

Linear Horizontal | Weighted w=2.0 | 51.385 | 0.51 | YES | 47

Linear Vertical | Euclidean | 50.941 | 5.86 | YES | 461

Linear Vertical | Manhattan | 50.941 | 7.60 | YES | 560

Linear Vertical | Chebyshev | 50.941 | 8.58 | YES | 690

Linear Vertical | Weighted w=0.5 | 50.941 | 17.11 | YES | 1346

Linear Vertical | Weighted w=1.0 | 50.941 | 5.94 | YES | 461

Linear Vertical | Weighted w=1.5 | 50.941 | 4.37 | YES | 306

Linear Vertical | Weighted w=2.0 | 59.527 | 3.19 | NO | 235

Linear Diagonal | Euclidean | 47.284 | 6.26 | YES | 458

Linear Diagonal | Manhattan | 47.284 | 0.52 | YES | 41

Linear Diagonal | Chebyshev | 47.284 | 8.37 | YES | 683

Linear Diagonal | Weighted w=0.5 | 47.284 | 17.02 | YES | 1368

Linear Diagonal | Weighted w=1.0 | 47.284 | 5.70 | YES | 458

Linear Diagonal | Weighted w=1.5 | 47.284 | 0.48 | YES | 40

Linear Diagonal | Weighted w=2.0 | 47.284 | 0.52 | YES | 40

Linear L-Shape | Euclidean | 48.698 | 0.91 | YES | 79

Linear L-Shape | Manhattan | 56.314 | 0.79 | NO | 66

Linear L-Shape | Chebyshev | 48.698 | 4.74 | YES | 382

Linear L-Shape | Weighted w=0.5 | 48.698 | 15.86 | YES | 1260

Linear L-Shape | Weighted w=1.0 | 48.698 | 0.90 | YES | 79

Linear L-Shape | Weighted w=1.5 | 48.698 | 0.48 | YES | 41

Linear L-Shape | Weighted w=2.0 | 48.698 | 0.47 | YES | 41

Geo Hollow Square | Euclidean | 48.456 | 3.65 | YES | 297

Geo Hollow Square | Manhattan | 53.728 | 0.66 | NO | 51

Geo Hollow Square | Chebyshev | 48.456 | 6.06 | YES | 504

Geo Hollow Square | Weighted w=0.5 | 48.456 | 14.05 | YES | 1148

Geo Hollow Square | Weighted w=1.0 | 48.456 | 3.70 | YES | 297

Geo Hollow Square | Weighted w=1.5 | 48.456 | 0.57 | YES | 51

Geo Hollow Square | Weighted w=2.0 | 49.284 | 0.58 | NO | 52

Geo Filled Circle | Euclidean | 47.284 | 2.72 | YES | 211

Geo Filled Circle | Manhattan | 52.799 | 0.80 | NO | 63

Geo Filled Circle | Chebyshev | 47.284 | 5.90 | YES | 474

Geo Filled Circle | Weighted w=0.5 | 47.284 | 13.85 | YES | 1131

Geo Filled Circle | Weighted w=1.0 | 47.284 | 2.59 | YES | 211

Geo Filled Circle | Weighted w=1.5 | 49.770 | 0.79 | NO | 65

Geo Filled Circle | Weighted w=2.0 | 50.598 | 0.69 | NO | 53

Geo Star | Euclidean | 49.627 | 5.75 | YES | 456

Geo Star | Manhattan | 55.728 | 2.07 | NO | 171

Geo Star | Chebyshev | 49.627 | 7.61 | YES | 632

Geo Star | Weighted w=0.5 | 49.627 | 15.62 | YES | 1301

Geo Star | Weighted w=1.0 | 49.627 | 5.84 | YES | 456

Geo Star | Weighted w=1.5 | 51.284 | 1.59 | NO | 129

Geo Star | Weighted w=2.0 | 53.770 | 1.84 | NO | 162

Geo Spiral | Euclidean | 49.770 | 5.93 | YES | 511

Geo Spiral | Manhattan | 49.770 | 1.58 | YES | 122

Geo Spiral | Chebyshev | 49.770 | 7.96 | YES | 710

Geo Spiral | Weighted w=0.5 | 49.770 | 16.27 | YES | 1385

Geo Spiral | Weighted w=1.0 | 49.770 | 5.88 | YES | 511

Geo Spiral | Weighted w=1.5 | 49.770 | 0.70 | YES | 54

Geo Spiral | Weighted w=2.0 | 50.355 | 0.61 | NO | 51

Maze Simple | Euclidean | 48.456 | 7.27 | YES | 565

Maze Simple | Manhattan | 48.456 | 0.53 | YES | 44

Maze Simple | Chebyshev | 48.456 | 8.76 | YES | 736

Maze Simple | Weighted w=0.5 | 48.456 | 17.25 | YES | 1423

Maze Simple | Weighted w=1.0 | 48.456 | 7.16 | YES | 565

Maze Simple | Weighted w=1.5 | 48.456 | 0.50 | YES | 42

Maze Simple | Weighted w=2.0 | 48.456 | 0.50 | YES | 42

Maze Rooms | Euclidean | 47.870 | 3.65 | YES | 285

Maze Rooms | Manhattan | 48.456 | 0.51 | NO | 42

Maze Rooms | Chebyshev | 47.870 | 6.10 | YES | 509

Maze Rooms | Weighted w=0.5 | 47.870 | 14.75 | YES | 1195

Maze Rooms | Weighted w=1.0 | 47.870 | 3.42 | YES | 285

Maze Rooms | Weighted w=1.5 | 48.456 | 0.51 | NO | 44

Maze Rooms | Weighted w=2.0 | 48.456 | 0.49 | NO | 42

Maze Open | Euclidean | 54.899 | 7.27 | YES | 781

Maze Open | Manhattan | 54.899 | 0.71 | YES | 78

Maze Open | Chebyshev | 54.899 | 7.83 | YES | 815

Maze Open | Weighted w=0.5 | 54.899 | 11.30 | YES | 1253

Maze Open | Weighted w=1.0 | 54.899 | 7.29 | YES | 781

Maze Open | Weighted w=1.5 | 54.899 | 0.51 | YES | 55

Maze Open | Weighted w=2.0 | 54.899 | 0.53 | YES | 53

Done.

Task 4

Here’s my Task 4 write-up, like I’d put in my report as a 25-year-old student 👇

**Pattern Impact Analysis (based on my benchmark table)**

**TL;DR**

* **Fastest / fewest nodes explored:** higher **Weighted Euclidean (w=1.5–2.0)** often explores the fewest nodes, but it sometimes gives **non-optimal** paths.
* **Safest + efficient:** **Manhattan** is super strong on grids and corridor-like patterns (low expansions, still optimal in many cases).
* **Most stable but rarely “best”:** **Euclidean** is consistently optimal and decent, especially around **curved** obstacles (circle, star).
* **Chebyshev:** rarely the winner here, but it’s solid when diagonal moves dominate; it stays optimal and isn’t too greedy.

**Which heuristic won (fewest explored) by pattern?**

* **Empty Grid:** Manhattan / Weighted 1.5–2.0 (≈ **40** nodes, optimal)
* **Simple Barrier:** Manhattan (~**78**, optimal). Weighted 1.5–2.0 explored fewer nodes but **not optimal**.
* **Maze DFS (tight corridors):** All about the same (~**498–508**). The maze forces lots of expansion no matter what.
* **Scattered Obstacles (20%):** Weighted **1.5** (≈ **55**, optimal). Weighted **2.0** was even smaller but **not optimal**.
* **Linear Horizontal:** Weighted **2.0** (~**47**, optimal).
* **Linear Vertical:** Best **optimal** was Weighted **1.5** (~**306**). Weighted **2.0** was smallest (~**235**) but **not optimal**.
* **Linear Diagonal:** Weighted **1.5** / Manhattan (~**40–41**, optimal).
* **Linear L-Shape:** Weighted **1.5–2.0** (~**41**, optimal). Manhattan was **not optimal** here.
* **Geo Hollow Square (hollow box):** Weighted **1.5** (~**51**, optimal). Manhattan had the same expansions but **not optimal**.
* **Geo Filled Circle:** **Euclidean** (~**211**, optimal). Weighted 1.5–2.0 went greedy and **lost optimality**.
* **Geo Star:** **Euclidean** / Weighted **1.0** (~**456**, optimal). Heavier weights + Manhattan cut corners and **failed optimality**.
* **Geo Spiral:** Weighted **1.5** (~**54**, optimal). Weighted **2.0** was smaller but **not optimal**.
* **Maze Simple:** Weighted **1.5–2.0** (~**42**, optimal).
* **Maze Rooms:** **Euclidean**/**Weighted 1.0** (~**285**, optimal). Manhattan and heavy weights were **not optimal**.
* **Maze Open (many paths):** Weighted **2.0** (~**53**, **still optimal** here). Nice case where strong greediness still worked.

**Patterns that caused the most exploring:**

* **Maze Open** and **large linear/diagonal layouts** with conservative heuristics (Euclidean/Chebyshev) led to **500–800+** nodes.
* **Tight DFS maze** equalized everything around ~500 nodes.

**Patterns with the least exploring:**

* **Straight-line friendly layouts** (Empty, Linear Diagonal/L-Shape) with **Manhattan** or **Weighted 1.5–2.0** (≈ **40–55** nodes).

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

* **Works better** when the map is basically **grid/corridor-like** and the shortest path is a mix of axis-aligned steps (even if diagonals are allowed). It gives a strong directional push with minimal “cone” of exploration. That’s why it crushed **Empty**, **Linear Diagonal**, **Simple/Maze-like corridors**.
* **Works worse** on **curvy or diagonal-friendly shapes** (circle, star, hollow square). With diagonals allowed, Manhattan can **overestimate** the true 8-way cost (it’s not the perfect admissible heuristic for diagonal movement), so A\* gets greedier in the wrong direction. In my results, Manhattan often ended **non-optimal** on those **geometric** patterns.

Plain English: Manhattan is awesome when the world looks like Manhattan streets. When the world looks like circles and stars, it guesses wrong.

**2) When would I prefer Chebyshev over Euclidean?**

* If I’m in a world where **diagonal movement is common and “king-distance” fits the geometry**, Chebyshev is a nice, simple heuristic. It essentially says “the cost is how far you are in the dominant axis,” which encourages diagonal steps.
* It’s useful when I want something **more diagonal-aware than Manhattan** but **less curved-biased than Euclidean**.
* In my runs it wasn’t the top dog, but it stayed **optimal** and behaved predictably in **mazes** and **gridy** layouts. I’d reach for it when diagonal motion is frequent and I don’t want the Euclidean bias around curves.

**3) What happens when I increase the heuristic weight?**

* **Higher weight ⇒ fewer nodes explored** (more greedy, faster).
* **But** after ~**w=1.5–2.0** I started seeing **non-optimal** paths on several patterns:
  + **Simple Barrier** (w≥1.5): non-optimal
  + **Scattered Obstacles** (w=2.0): non-optimal
  + **Geometric shapes** (circle, star, hollow square): heavy weights often **non-optimal**
* Some maps (like **Maze Open**) still stayed optimal at **w=2.0**, but that’s a bit lucky—structure helped.

So, **weight is a trade-off**: crank it up when you need speed and can live with small sub-optimality; keep it **≤1.0** (or use Manhattan/Euclidean) when you need guaranteed optimal paths.

**My quick takeaways**

* If I need **optimal + decent speed**, I’ll use **Euclidean** (or **Manhattan** with diagonals OFF).
* If I need **speed** and can tolerate slight sub-optimality, I’ll bump **Weighted Euclidean to ~1.5** (maybe 2.0 if the map is friendly, like Maze Open).
* **Chebyshev** is a reasonable middle-ground when diagonals are common, but it wasn’t the winner in my shapes.
* Manhattan shines in **grid corridors**; it’s not great around **curved geometry**.

Part 4 assignment

**Part 1 – Random action control**

**A graph of a number of episodes

Description automatically generated**

**Command:**

python cartpole\_search.py --task random --episodes 1000 --plot random\_hist.png

**Result:**

* Average reward ≈ **22.4**
* Histogram saved: random\_hist.png

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**

**A graph with blue lines and a blue line

Description automatically generated**

**Command:**

python cartpole\_search.py --task angle --episodes 1000 --plot angle\_hist.png

**Result:**

* Average reward ≈ **41.9**
* Histogram saved: angle\_hist.png

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 optimization**

**A graph of black and red dots

Description automatically generated**

**Command:**

python cartpole\_search.py --task random\_search --iters 1000 --per\_iter 20 --scatter search\_scatter.png

**Result:**

* Best weights = [ 0.1167, 0.2187, 0.8714, 0.2236 ]
* Best average = **500.0** (maximum possible in CartPole-v1)
* 3D scatter: search\_scatter.png
* Top-3 important features: indices [3, 2, 1] → roughly **(theta\_dot, theta, x\_dot)**

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.

Question 4

Report the best weights + 1000-ep average and the histogram.

zsh: command not found: UZ39OQROE

(/Users/pascal-maker/pascalworkspace/env) pascal-maker@MacBook-Pro-van-Pascal reinforcementlearning % python cartpole\_search.py --task hill --iters 1000 --per\_iter 20 --sigma 0.1 --plot hill\_hist.png

[Hill] iter 50/1000 cur=500.0 best=500.0

[Hill] iter 100/1000 cur=500.0 best=500.0

[Hill] iter 150/1000 cur=500.0 best=500.0

[Hill] iter 200/1000 cur=500.0 best=500.0

[Hill] iter 250/1000 cur=500.0 best=500.0

[Hill] iter 300/1000 cur=500.0 best=500.0

[Hill] iter 350/1000 cur=500.0 best=500.0

[Hill] iter 400/1000 cur=500.0 best=500.0

[Hill] iter 450/1000 cur=500.0 best=500.0

[Hill] iter 500/1000 cur=500.0 best=500.0

[Hill] iter 550/1000 cur=500.0 best=500.0

[Hill] iter 600/1000 cur=500.0 best=500.0

[Hill] iter 650/1000 cur=500.0 best=500.0

[Hill] iter 700/1000 cur=500.0 best=500.0

[Hill] iter 750/1000 cur=500.0 best=500.0

[Hill] iter 800/1000 cur=500.0 best=500.0

[Hill] iter 850/1000 cur=500.0 best=500.0

[Hill] iter 900/1000 cur=500.0 best=500.0

[Hill] iter 950/1000 cur=500.0 best=500.0

[Hill] iter 1000/1000 cur=500.0 best=500.0

[HillClimb] Best weights: [ 0.1876 -0.0255 2.3374 0.6063]

[HillClimb] Test average reward (1000 episodes): 500.00

[saved] hill\_hist.png

A graph with numbers and a bar

Description automatically generated

Question 6

[saved] hill\_hist.png

(/Users/pascal-maker/pascalworkspace/env) pascal-maker@MacBook-Pro-van-Pascal reinforcementlearning % python cartpole\_search.py --task anneal --iters 1000 --per\_iter 20 --sigma 0.1 --t0 1.0 --cool 0.995 --adaptive --plot sa\_hist.png

[SA(adaptive)] iter 50/1000 cur=9.2 best=9.2 T=0.7783 sigma=0.0001

[SA(adaptive)] iter 100/1000 cur=9.2 best=9.2 T=0.6058 sigma=0.0001

[SA(adaptive)] iter 150/1000 cur=9.2 best=9.2 T=0.4715 sigma=0.0001

[SA(adaptive)] iter 200/1000 cur=9.2 best=9.2 T=0.3670 sigma=0.0001

[SA(adaptive)] iter 250/1000 cur=9.2 best=9.2 T=0.2856 sigma=0.0001

[SA(adaptive)] iter 300/1000 cur=9.2 best=9.2 T=0.2223 sigma=0.0001

[SA(adaptive)] iter 350/1000 cur=9.2 best=9.2 T=0.1730 sigma=0.0001

[SA(adaptive)] iter 400/1000 cur=9.2 best=9.2 T=0.1347 sigma=0.0001

[SA(adaptive)] iter 450/1000 cur=9.2 best=9.2 T=0.1048 sigma=0.0001

[SA(adaptive)] iter 500/1000 cur=9.2 best=9.2 T=0.0816 sigma=0.0001

[SA(adaptive)] iter 550/1000 cur=9.2 best=9.2 T=0.0635 sigma=0.0001

[SA(adaptive)] iter 600/1000 cur=9.2 best=9.2 T=0.0494 sigma=0.0001

[SA(adaptive)] iter 650/1000 cur=9.2 best=9.2 T=0.0385 sigma=0.0001

[SA(adaptive)] iter 700/1000 cur=9.2 best=9.2 T=0.0299 sigma=0.0001

[SA(adaptive)] iter 750/1000 cur=9.2 best=9.2 T=0.0233 sigma=0.0001

[SA(adaptive)] iter 800/1000 cur=9.2 best=9.2 T=0.0181 sigma=0.0001

[SA(adaptive)] iter 850/1000 cur=9.2 best=9.2 T=0.0141 sigma=0.0001

[SA(adaptive)] iter 900/1000 cur=9.2 best=9.2 T=0.0110 sigma=0.0001

[SA(adaptive)] iter 950/1000 cur=9.2 best=9.2 T=0.0085 sigma=0.0001

[SA(adaptive)] iter 1000/1000 cur=9.2 best=9.2 T=0.0067 sigma=0.0001

[SimAnneal (adaptive)] Best weights: [ 0.0012 0.2987 -0.2741 -0.8906]

[SimAnneal (adaptive)] Test average reward (1000 episodes): 9.15

[saved] sa\_hist.png

**Hill Climbing Results**

**Command used**

python cartpole\_search.py --task hill --iters 1000 --per\_iter 20 --sigma 0.1 --plot hill\_hist.png

**Output summary**

Best weights: [ 0.1876 -0.0255 2.3374 0.6063]

Average reward (1000 episodes): 500.00

The hill-climbing algorithm reached the **maximum possible reward (500)** on average.  
This means it **perfectly balanced the pole** for almost all 1000 episodes.

The best weight vector:

* **w₀ = 0.1876** → relates to the **cart position** (moderate effect).
* **w₁ = −0.0255** → relates to **cart velocity** (minor effect).
* **w₂ = 2.3374** → relates to **pole angle** (strongest weight).
* **w₃ = 0.6063** → relates to **pole angular velocity** (important stabilizer).

The large positive value on w₂ shows that **pole angle** is the most critical factor: when the pole starts tilting, this weight drives the corrective action.

**Interpretation:**  
Hill climbing worked really well because once it found a stable set of weights, it stuck with them. The environment is deterministic enough that small tweaks weren’t needed once the solution was near-perfect.

The histogram (hill\_hist.png) should show almost all episodes near **500 timesteps**, confirming consistent stability.

**Simulated Annealing (Adaptive) Results**

**Command used**

python cartpole\_search.py --task anneal --iters 1000 --per\_iter 20 --sigma 0.1 --t0 1.0 --cool 0.995 --adaptive --plot sa\_hist.png

**Output summary**

Best weights: [ 0.0012 0.2987 -0.2741 -0.8906]

Average reward (1000 episodes): 9.15

⚠️ The simulated annealing version **did not converge** to a good policy — the reward stayed around **9 timesteps**, which is almost random behavior.

**Possible reasons:**

* The **adaptive noise** got too small (sigma = 0.0001), preventing meaningful exploration.
* The **temperature** (T) cooled too quickly, reducing the chance of accepting better random moves.
* The **initial weights** might have been unlucky, and the algorithm “froze” early.

**Interpretation:**  
Simulated annealing explores widely at first, then narrows down — but here it cooled too fast and got “stuck.”  
The low reward means it failed to stabilize the pole for long; the policy wasn’t strong enough to react correctly to pole tilts.

**Comparison & Conclusion**

| **Algorithm** | **Avg Reward** | **Performance** | **Notes** |
| --- | --- | --- | --- |
| Random | ~22 | Poor | Random jerky movements |
| Angle-based | ~42 | Better | Some reactive logic but unstable |
| Random Search | 500 | Excellent | Found optimal weights by brute force |
| Hill Climbing | 500 | Excellent | Converged efficiently to stable control |
| Simulated Annealing | ~9 | Failed | Over-cooled; got stuck early |

**Summary in plain words (for your report):**

“Hill climbing and random search both managed to perfectly balance the pole, achieving an average reward of 500. The best-performing weights strongly emphasize the pole angle and angular velocity, showing these are the key features for stability. Simulated annealing underperformed, likely due to aggressive cooling and vanishing noise, which prevented proper exploration. In future experiments, I’d use a slower cooling rate or fixed noise to maintain diversity.”

A graph with a line

Description automatically generated

**Brief answers for 1.6 (discussion prompts)**

* **Reward vs. compute:**
  + *Random search* explores broadly but wastes samples; good baseline.
  + *Hill climbing* is cheap and can quickly find decent weights, but may get stuck.
  + *Simulated annealing* is hill climbing with occasional downhill moves; better at escaping local optima; with **adaptive noise**, it self-tunes step sizes (often converges faster/steadier).
* **Feature expansion:**  
  Yes, it can help. Typical useful features for CartPole: products and squares like theta, theta\_dot, x, x\_dot, theta\*x, theta\_dot\*x, theta^2, theta\_dot^2. You’d expand obs and grow w to match. (Keep the linear decision rule.)
* **MountainCar-v0** (quick strategy):
  + Purely linear policies struggle because reward comes only at the terminal.
  + Two effective search-based options:
    1. **Simulated annealing / hill climb** on a **feature-expanded** state (e.g., tile coding / RBFs / polynomial features of position & velocity), choosing action by sign of w·phi(s).
    2. **Reward shaping**: add velocity bonus or shaping potential (e.g., +c·|v|) to guide search.
  + Also consider **episode length** and **noise scaling**: big steps early (explore), smaller later (exploit).