## Appendix A: Creative Agent Implementations

## 1. Word Discovery Task Agents

All agents operate under the same feedback: match\_score(x,target) = number of exact character–position matches. Step budgets (T\_max) are identical across agents for fair comparison.

### 1.1. TransBot Agent

Purpose: Improve the best-known string using periodic structure-level transforms (reverse/shift) and otherwise single-character mutations; accept only strict improvements.

Inputs:

* target : string
* alphabet : set
* match\_score(x, target) → int

Parameters:

* T\_max : int // step budget
* transform\_period : int = 10 // apply a transform every k steps
* transform\_set := {reverse, shift\_left, shift\_right }

State:

* best ← sample\_uniform\_string(L, alphabet)
* best\_score ← match\_score(best, target)

Procedure:

|  |
| --- |
| 1. For t = 1 … T\_max: |
| a. y ← best |
| b. If t mod transform\_period = 0: |
| i. τ ← random\_choice(transform\_set) |
| ii. If τ = reverse: y ← reverse(y) |
| Else if τ = shift\_left: |
| y ← y[2..L] • random\_char(alphabet) // left shift; append random |
| Else if τ = shift\_right: |
| y ← random\_char(alphabet) • y[1..L-1] // right shift; prepend random |
| Else: |
| i. i\* ← random\_int(1, L) |
| ii. y[i\*] ← random\_char(alphabet) // point mutation |
| c. s ← match\_score(y, target) |
| d. If s > best\_score: |
| i. best ← y |
| ii. best\_score ← s |
| 2. Return best\_score |

### 1.2. ComboBot Agent

Purpose: Maintain a pool of improved strings; generate children by two-parent single-cut splicing (or fallback mutation when pool is small); accept only strict improvements and expand pool.

Inputs:

* target : string
* alphabet : set
* match\_score(x, target) → int

Parameters:

* T\_max : int
* p\_mutate : float = 0.3 // probability to apply a single-point mutation to child

State:

* best ← sample\_uniform\_string(L, alphabet)
* best\_score ← match\_score(best, target)
* pool ← [best]

Procedure:

|  |
| --- |
| 1. For t = 1 … T\_max: |
| a. If |pool| ≥ 2: |
| i. (p1, p2) ← random\_sample(pool, 2) |
| ii. split ← random\_int(1, L−1) |
| iii. child ← p1[1..split] • p2[split+1..L] // one-point crossover |
| Else: |
| i. child ← best |
| ii. i\* ← random\_int(1, L) |
| iii. child[i\*] ← random\_char(alphabet) // fallback mutation |
| b. With probability p\_mutate: |
| i. j\* ← random\_int(1, L) |
| ii. child[j\*] ← random\_char(alphabet) |
| c. s ← match\_score(child, target) |
| d. If s > best\_score: |
| i. best ← child |
| ii. best\_score ← s |
| iii. append(pool, child) |
| 2. Return best\_score |

### 1.3. InferoBot Agent

Purpose: Maintain independent categorical distributions over characters per position; sample from current distributions and blindly reinforce character–position counts whenever a newly sampled candidate strictly improves the best score.

Inputs:

* target : string
* alphabet : ordered list = [a1, …, aM]
* match\_score(x, target) → int

Parameters:

* T\_max : int
* α : float = 1.0 // symmetric Dirichlet-like prior per position

State:

* For each position i ∈ {1..L}: counts[i] ← [α, α, …, α] (length M) // initial uniform pseudo-counts
* best\_score ← 0
* best ← null

Procedure:

|  |
| --- |
| 1. For t = 1 … T\_max: |
| a. For i = 1..L: |
| i. p\_i ← normalize(counts[i]) // convert counts to probabilities |
| ii. c\_i ← sample\_from(alphabet, p\_i) |
| b. x ← concatenate(c\_1, …, c\_L) |
| c. s ← match\_score(x, target) |
| d. If s > best\_score: |
| i. best\_score ← s |
| ii. best ← x |
| iii. For i = 1..L: // blind reinforcement of used tokens |
| idx ← index\_of(x[i], alphabet) |
| counts[i][idx] ← counts[i][idx] + 1 |
| 2. Return best\_score |

## 2. Maze Navigation Task Agents

All maze agents use the same grid world, step budget, and success criterion (reaching goal).

### 2.1. TransBot Agent

Purpose: Every k steps, transform the entire path (reverse / rotate); otherwise step randomly.

Inputs:

* maze, start, goal
* T\_max
* transform\_period k = 10
* transform\_set = {reverse, rotate\_left, rotate\_right }

State:

* path ← [start]
* visited ← {start}

Procedure:

|  |
| --- |
| 1. For t = 1 … T\_max: |
| a. If last(path) == goal: return (True, |path|) |
| b. If t mod k = 0 and |path| > 1: |
| i. τ ← random\_choice(transform\_set) |
| ii. If τ = reverse: path ← reverse(path) |
| Else if τ = rotate\_left: |
| path ← path[2..|path|] • path[1] // move head to tail |
| Else if τ = rotate\_right: |
| path ← last(path) • path[1..|path|−1] // move tail to head |
| c. current ← last(path) |
| d. N ← { n ∈ get\_neighbors(current) : n ∉ visited } |
| e. If N = ∅: break |
| f. step ← random\_choice(N) |
| g. append(path, step); visited ← visited ∪ {step} |
| 1. For t = 1 … T\_max: |
| a. If last(path) == goal: return (True, |path|) |
| b. If t mod splice\_period = 0 and fragments ≠ []: |
| i. frag ← random\_choice(fragments) |
| ii. For f in frag (in order): |
| If f is open in maze AND f ∉ visited: |
| append(path, f); visited ← visited ∪ {f}; goto (d) |
| c. Else: |
| i. current ← last(path) |
| ii. N ← { n ∈ get\_neighbors(current) : n ∉ visited } |
| iii. If N = ∅: break |
| iv. step ← random\_choice(N) |
| v. append(path, step); visited ← visited ∪ {step} |
| d. If t mod fragment\_period = 0: |
| i. frag ← last\_k(path, fragment\_length) // e.g., last 3 coords |
| ii. append(fragments, frag) |
| 2. Return (False, |path|) |

### 2.2. ComboBot Agent

Purpose: Periodically store recent path fragments; sometimes splice a stored fragment by appending its first valid, unvisited cell; otherwise step randomly.

Inputs:

* maze, start, goal
* T\_max
* fragment\_period = 15
* splice\_period = 10
* fragment\_length ≈ 3 // last 3 states

State:

* path ← [start]
* fragments ← []
* visited ← {start}

Procedure:

|  |
| --- |
| 1. For t = 1 … T\_max: |
| a. If last(path) == goal: return (True, |path|) |
| b. If t mod splice\_period = 0 and fragments ≠ []: |
| i. frag ← random\_choice(fragments) |
| ii. For f in frag (in order): |
| If f is open in maze AND f ∉ visited: |
| append(path, f); visited ← visited ∪ {f}; goto (d) |
| c. Else: |
| i. current ← last(path) |
| ii. N ← { n ∈ get\_neighbors(current) : n ∉ visited } |
| iii. If N = ∅: break |
| iv. step ← random\_choice(N) |
| v. append(path, step); visited ← visited ∪ {step} |
| d. If t mod fragment\_period = 0: |
| i. frag ← last\_k(path, fragment\_length) // e.g., last 3 coords |
| ii. append(fragments, frag) |
| 2. Return (False, |path|) |

### 2.3. InferoBot Agent

Purpose: Maintain a categorical prior over open cells; each step, sample a target cell from priors, move one random step from current, and if that move reduces best distance-to-goal, accept it and (under a condition) reinforce the chosen next cell.

Inputs:

* maze, start, goal
* T\_max

Precompute:

* O ← list of all open cells in maze
* index : O → {1..|O|}

State:

* α ← 1.0
* priors ← [α, α, …, α] over O // uniform
* best\_dist ← +∞
* steps\_accepted ← 0
* current ← start

Procedure:

|  |
| --- |
| 1. For t = 1 … T\_max: |
| a. idx\_tgt ← sample\_categorical(normalize(priors)) |
| b. target ← O[idx\_tgt] // (used only as a bias signal) |
| c. N ← get\_neighbors(current) |
| d. If N = ∅: break |
| e. cand ← random\_choice(N) // fairness: random neighbor |
| f. d ← euclidean(cand, goal) |
| g. If d < best\_dist: // improvement-only acceptance |
| i. best\_dist ← d |
| ii. current ← cand |
| iii. steps\_accepted ← steps\_accepted + 1 |
| h. If euclidean(current, goal) < euclidean(start, goal): |
| i. priors[index[cand]] ← priors[index[cand]] + 1 // reinforce cand cell |
| i. If current == goal: return (True, steps\_accepted) |
| 2. Return (False, steps\_accepted) |

### 2.4. ExploreBot Agent

Purpose: Prefer neighbors least visited so far (novelty heuristic); break ties arbitrarily.

Inputs:

* maze, start, goal
* T\_max

State:

* path ← [start]
* visit\_count : map cell → int, default 0
* visit\_count[start] ← 1

Procedure:

|  |
| --- |
| 1. For t = 1 … T\_max: |
| a. current ← last(path) |
| b. If current == goal: return (True, |path|) |
| c. N ← get\_neighbors(current) |
| d. If N = ∅: break |
| e. next ← argmin\_{n∈N} visit\_count[n] // choose least-visited neighbor |
| f. append(path, next) |
| g. visit\_count[next] ← visit\_count[next] + 1 |
| 2. Return (False, |path|) |