Универзитет Црне Горе

Природно-математички факултет

Математички софтверски пакети

Примјена Active Inference-а на проблем Cliff Walking

Студент:
Никола Поповић 1/24 Б

Ментор:
мр Никола Пижурица

мај 2025.

Садржај

Увод

У овом раду представљена је имплементација агента заснованог на принципу Active Inference-а у окружењу Cliff Walking, доступном у оквиру библиотеке Gymnasium. Циљ агента је да научи како да безбједно дође до циљног поља, избјегавајући опасне провалије, ослањајући се искључиво на свој генеративни модел свијета и принцип минимизације очекиване слободне енергије.

У оквиру рада реализована је имплементација агента који не учи структуру свијета кроз интеракцију, већ почиње са унапријед дефинисаним генеративним моделом: матрицама A (вјероватноћа запажања у зависности од скривених стања), B (матрица транзиција стања), C (преференције) и D (почетна увјерења). Агент планира низ акција (политике) и бира ону која минимизује очекивану слободну енергију.

Поред основне верзије агента, дата је имплементација у још два сценарија:

1. када је додат шум у матрицама A и D, и
2. када окружење постаје стохастичко (укључивањем is_slippery параметра).

1 Опис проблема и окружења

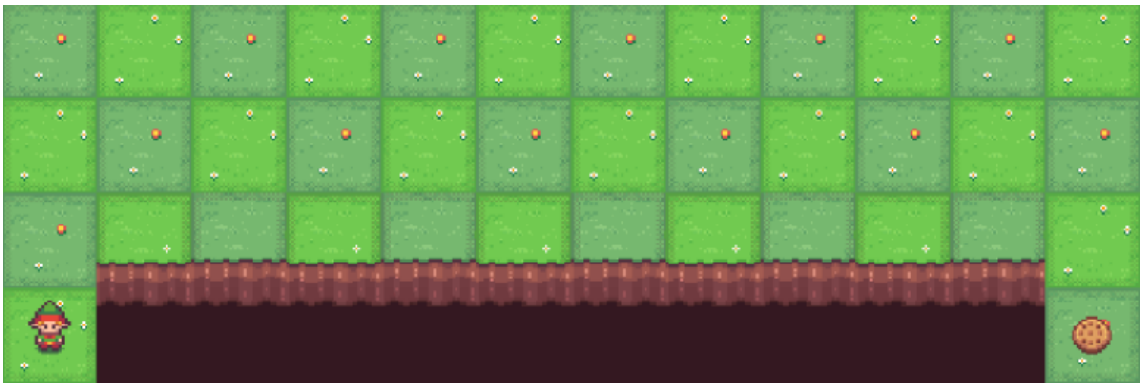
Окружење представља мрежу величине 4×12, у којој агент почиње у позицији $[3, 0]$, односно у доњем лијевом углу. Циљ агента је да достигне позицију $[3, 11]$, која се налази у доњем десном углу мреже. Када агент достигне циљ, епизода се завршава.

Дуж реда $[3, 1]$ до $[3, 10]$ налази се провалија. Ако агент крочи на неко од ових поља, одмах се враћа на почетну позицију. Агент извршава акције док не достигне циљ, али уколико направи грешку и упадне у провалију, мора поново да крене од почетка.

Простор акција агента састоји се од четири могућа покрета: помјерања горе, десно, доље или лијево, које су означене бројевима од 0 до 3. При томе, постоји могућност да окружење буде клизаво (is_slippery режим, подразумијевано искључен), што значи да агент понекад може да се помјери у неком другом смјеру, који се налази нормално у односу на онај који је наумио.

Сваки корак агента носи награду од -1, док је казна за корак у провалију значајно већа и износи -100. Епизода се завршава у тренутку када агент стигне на циљну позицију.

Простор опсервација садржи $3 \times 12 + 1 = 37$ могућих стања. Агент се не може налазити на провалији, нити на циљну позицију јер долазак на циљ прекида епизоду. Тренутна позиција агента представљена је као један цио број, израчунат по формули: тренутни_ред $\times$ број_колона + тренутна_колона, гдје индексирање реда и колона креће од 0. На примјер, почетна позиција $[3, 0]$ одговара индексу $3 \times 12 + 0 = 36$.



Слика 1.1: Почетно стање агента у Cliff Walking окружењу

2 Имплементација Active Inference агента

2.1 Стандардни модел

У овој фази рада фокусираћемо се на класичан модел окружења за Active Inference агента, где су матрице вјероватноћа запажања у зависности од скривених стања и почетних уверења детерминистичке, односно без шумова. То значи да матрица A, која описује вероватноћу посматрања датог стања, и матрица D, која представља агентову почетну претпоставку о свом положају, садрже само сигурне и јасне вриједности без случајних варијација.

Поред тога, параметар клизавог окружења is_slippery је (подразумијевано) у овом моделу искључен, што значи да се агент помјера строго у правцу изабране акције.

2.1.1 Учитавање библиотека. Визуелизација окружења и увјерења.

Испод је приказан Python код за учитавање потребних библиотека:

```
import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import itertools
from pymdp import utils
from pymdp.maths import softmax, spm_log_single as log_stable
from pymdp.control import construct_policies
```

Код 2.1.1: Учитавање библиотека

За боље разумијевање процеса учења и понашања агента, имплементиране су функције за визуализацију. Оне су преузете из Active inference from scratch и димензије су прилагођене нашем проблему.

```
def plot_likelihood(matrix, xlabels=None, ylabels=None,
                    title_str="Likelihood distribution (A)"):
    """
    Plots a 2-D likelihood matrix as a square heatmap with a controlled
    plot size.
    """
    if not np.isclose(matrix.sum(axis=0), 1.0).all():
        raise ValueError("Distribution not column-normalized!
                         Please normalize
```

```
                        (ensure matrix.sum(axis=0) == 1.0 for all
                        columns)")

    if xlabels is None:
        xlabels = list(range(matrix.shape[1]))
    if ylabels is None:
        ylabels = list(range(matrix.shape[0]))

    # Adjust the figure size manually for more controlled sizing
    max_size = 8  # Maximum figure size (in inches)
    size = max(matrix.shape)  # Get the larger of (rows, cols)

    # Use a fixed figure size to prevent the plot from being too big
    # Scale the figure size based on the matrix shape but constrain the
    size
    fig_size = (min(size, max_size), min(size, max_size))

    plt.figure(figsize=fig_size)  # Set the figure size

    sns.heatmap(matrix, xticklabels=xlabels, yticklabels=ylabels,
    cmap='gray', cbar=False, vmin=0.0, vmax=1.0, square=True)

    plt.title(title_str)

    plt.show()


def plot_grid(grid_locations, num_x=4, num_y=12):
    """
    Plots the spatial coordinates of GridWorld as a heatmap, with each
   (X, Y) coordinate labeled with its linear index (its `state id`).
    """
    grid_heatmap = np.zeros((num_x, num_y))

    for linear_idx, location in enumerate(grid_locations):
      y, x = location
      grid_heatmap[y, x] = linear_idx

    sns.set(font_scale=1.5)

    sns.heatmap(grid_heatmap, annot=True, cbar = False,
    fmt='.0f', cmap='crest')

    plt.show()
def plot_point_on_grid(state_vector, grid_locations):
```

```
    """
    Plots the current location of the agent on the grid world
    """

    state_index = np.where(state_vector)[0][0]

    y, x = grid_locations[state_index]

    grid_heatmap = np.zeros((4,12))

    grid_heatmap[y,x] = 1.0

    sns.heatmap(grid_heatmap, cbar = False, fmt='.0f')

    plt.show()


def plot_beliefs(belief_dist, title_str="Belief Distribution"):
    """
    Plot a categorical distribution or belief distribution, stored in
    the 1-D numpy vector `belief_dist`.
    """

    if not np.isclose(belief_dist.sum(), 1.0):
        raise ValueError("Distribution not normalized!
                         Please normalize")


    plt.figure(figsize=(12, 4))

    plt.grid(zorder=0)

    plt.bar(range(belief_dist.shape[0]), belief_dist, color='r',
    zorder=3)

    plt.xticks(range(belief_dist.shape[0]))

    plt.title(title_str)

    plt.show()
```

Код 2.1.2: Функције за цртање графика

2.1.2 Иницијализација окружења. Дефинисање основних параметара и структуре стања

У овом дијелу креирамо окружење „CliffWalking-v0“ из библиотеке Gymnasium са режимом приказа који враћа RGB низ пиксела. Функција display_frame служи за приказивање тренутног стања окружења као слике, гдје се искључују осе и додаје наслов ради боље визуализације, а команда plt.show(block = True) осигурава да се прозор са сликом приказује интерактивно и да се програм не настави док корисник не затвори тај прозор.

```
env = gym.make("CliffWalking-v0", render_mode="rgb_array")
obs, info = env.reset()

def display_frame(frame, title_str=""):
  plt.figure(figsize=(10,4))
  plt.imshow(frame)
  plt.axis("off")
  plt.title(title_str)
  plt.show(block=True)
```

Код 2.1.3: Инизијализација и функција за приказ окружења

Овдје је дат код који описује структуру мреже стања (grid world), број стања и опажања, као и скуп акција које агент може предузети. Ове основне компоненте представљају темељ за изградњу модела и даљу симулацију.

```
grid_locations = list(itertools.product(range(4), range(12)))

n_states = len(grid_locations)
n_observations = len(grid_locations)

actions = ["UP", "RIGHT", "DOWN", "LEFT"]
```

Код 2.1.4: Дефинисање стања, опажања и акција

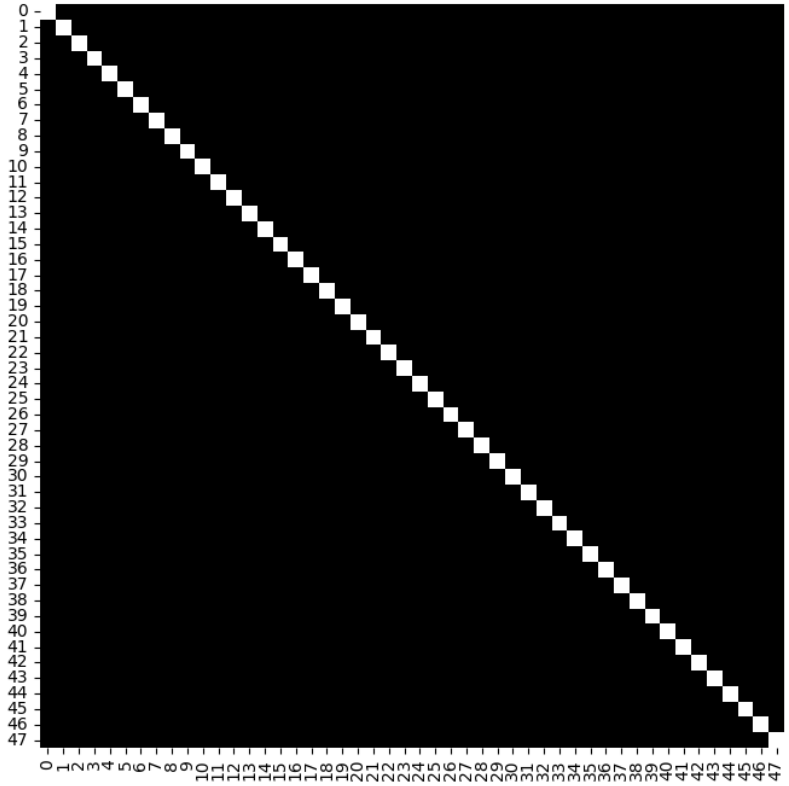
2.1.3 Генеративни модел

У наставку формулишемо и описујемо основне компоненте генеративног модела које карактеришу понашање окружења и унутрашња уверења агента.

Матрица A представља условне вероватноће опсервација у зависности од тренутног стања агента. Када је матрица A јединична, то значи да свака опсервација јасно и директно одговара једном стању, без икакве неодређености или шума. Другим ријечима, ако се агент налази у стању s_i, он са сигурношћу 1 опсервира стање o_i. Оваква поставка поједностављује модел, јер нема нејасноћа у посматрањима — свака позиција је прецизно идентификована.

```
A = np.eye(n_observations, n_states)

plot_likelihood(A)
```

Код 2.1.5: Дефинисање матрице A и њен графички приказ



Слика 2.1: График вјероватноћа запажања у зависности од скривених стања

Матрица транзиције B дефинише како се стања мијењају као резултат извршене акције. За сваку акцију и тренутно стање, ова матрица садржи вјероватноће преласка у следеће стање. Ако акција води ван граница дефинисане мреже, позиција остаје непромијењена. Посебан случај представљају провалије на дну мреже, гдје је агент уколико ступи у ту зону враћен на почетну позицију. Важно је напоменути да су у овом моделу преласци потпуно детерминистички: ако агент одабере неку акцију, она ће засигурно бити извршена, осим ако не води ван оквира мреже.

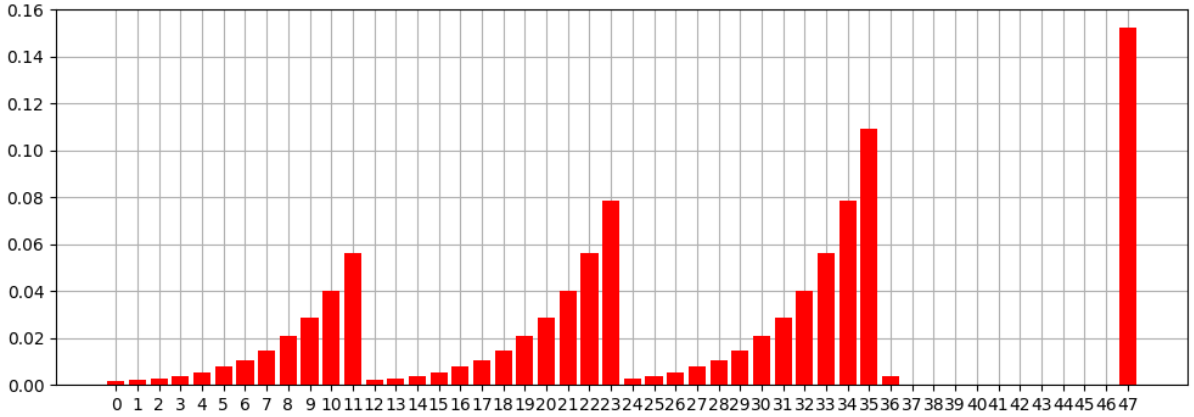
```
def create_B_matrix():
  B = np.zeros( (len(grid_locations), len(grid_locations),
                 len(actions)) )

  for action_id, action_label in enumerate(actions):

    for curr_state, grid_location in enumerate(grid_locations):

      y, x = grid_location

      if (y == 3 and 1 <= x <= 10):
        x = 0
        y = 3

      if action_label == "UP":
        next_y = y - 1 if y > 0 else y
        next_x = x
      elif action_label == "DOWN":
        next_y = y + 1 if y < 3 else y
        next_x = x
      elif action_label == "LEFT":
        next_x = x - 1 if x > 0 else x
        next_y = y
      elif action_label == "RIGHT":
        next_x = x + 1 if x < 11 else x
        next_y = y

      new_location = (next_y, next_x)
      next_state = grid_locations.index(new_location)
      B[next_state, curr_state, action_id] = 1.0
  return B


B = create_B_matrix()
```

Код 2.1.6: Дефинисање матрице B

Вектор преференција C одражава жељена стања, односно циљеве агента. Вриједности су засноване на негативној скалираној Менхетн удаљености од циљне позиције — што је неко стање даље од циља, то је мање пожељно. Конкретно, удаљеност се дијели са 3 да би се ублажила разлика између стања. Провалије имају изузетно ниске вриједности, чиме се избјегавају као непожељне. Након инизијализације, C се нормализује коришћењем softmax функције, добијајући вјероватосну дистрибуцију коју агент користи приликом планирања.

```
C = np.full(n_states, -1.0)

for idx in range(n_states):
    i, j = grid_locations[idx]
    distance = abs(3 - i) + abs(11 - j)
    c_value = -distance / 3

    if i == 3 and 1 <= j <= 10:
        c_value = -100
    C[idx] = c_value

C = softmax(C)


plot_beliefs(C)
```

Код 2.1.7: Дефинисање матрице C и њен графички приказ

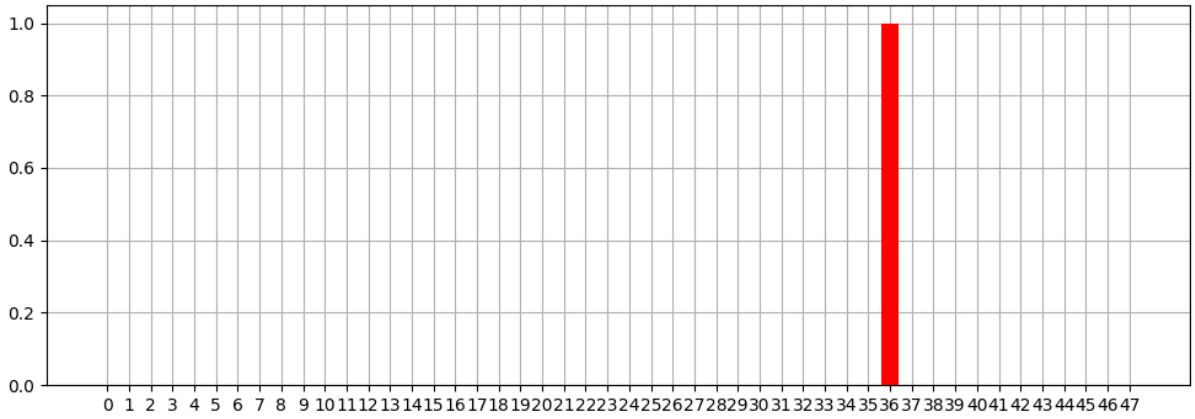


Слика 2.2: График пожељних стања

Почетна уверења D представљају почетно стање у којем агент верује да се налази, што је у овом случају локација $[3, 0]$. То је моделовано као јединични вектор (one-hot) који има вриједност 1 на почетном стању и 0 на свим осталим.

```
D = utils.onehot(grid_locations.index((3,0)), n_states)

plot_beliefs(D)
```

Код 2.1.8: Дефинисање матрице D и њен графички приказ



Слика 2.3: График вјероватноћа почетних стања

2.1.4 Функције за инференцију и рачунање очекиване слободне енергије

Након дефинисања генеративног модела, следећи корак јесте доношење закључака о скривеним стањима на основу опсервација и претходних увјерења, као и предвиђање последица различитих потенцијалних акција. На овом мјесту представљене су двије суштинске компоненте Active Inference-а: ажурирање увјерења, којом агент изводи претпоставке о свом тренутном стању, и планирање путем очекиване слободне енергије, која омогућава агенту да процијени и изабере најпожељније акције на основу предвиђених исхода. Ове компоненте заједно чине основу интелигентног понашања заснованог на принципу минимизације слободне енергије.

Ове функције су преузете из Active Inference from Scratch и прилагођене нашем задатку. Тамо се може прочитати нешто више о њима.

Функција infer_states ажурира увјерење агента о скривеним стањима на основу нове опсервације и претходног увјерења, користећи Бајесово правило у логаритамском простору. Конкретно, логаритам условне вероватноће запажања у зависности од скривеног стања и логаритам дистрибуције претходног увјерења сабирају се елемент по елемент. Тако добијени збир се затим нормализује примјеном softmax функције.

```
def infer_states(observation_index, A, prior):
  log_likelihood = log_stable(A[observation_index,:])

  log_prior = log_stable(prior)

  qs = softmax(log_likelihood + log_prior)

  return qs
```

Код 2.1.9: Функција infer_states

Сада ћемо имплементирати функцију која предвиђа која ће скривена стања вероватно настати један корак у будућност, ако се изврши одређена акција.

```
def get_expected_states(B, qs_current, action):
  qs_u = B[:,:,action].dot(qs_current)

  return qs_u
```

Код 2.1.10: Функција get_expected_states

Наредна функција предвиђа која ће опажања највероватније бити добијена након што агент уђе у одређена стања.

```
def get_expected_observations(A, qs_u):
  qo_u = A.dot(qs_u)

  return qo_u
```

Код 2.1.11: Функција get_expected_observations

Дајемо сада имплементацију функција entropy и kl_divergence. Прва израчунава ентропију (мјеру неизвјесности) за свако скривено стање, на основу вјероватноћа различитих опажања у том стању. Друга израчунава Кулбак-Лајблерову дивергенцију између двије вјероватносне дистрибуције: qo_u, очекивана опажања (шта агент

очекује да види) и C, агентове претходне преференције (шта жели да види). Она мјери колико се очекивана опажања разликују од онога што агент жели.

```
def entropy(A):
  H_A = - (A * log_stable(A)).sum(axis=0)

  return H_A


def kl_divergence(qo_u, C):
  return (log_stable(qo_u) - log_stable(C)).dot(qo_u)
```

Код 2.1.12: Функције entropy и kl_divergence

Следећа функција представља централни механизам планирања у оквиру Active Inference-а. Она израчунава очекивану слободну енергију (G) за сваки могући низ акција, односно политику. За сваку политику симулира како би се увјерење о скривеним стањима и очекиваним опажањима развијало током предвиђеног временског хоризонта.

Очекивана слободна енергија се састоји од два главна члана:

1. Очекивана ентропија опажања — мјера неизвјесности коју агент предвиђа да ће искусити.
2. Кулбак-Лајблерова дивергенција између предвиђених и пожељних опажања — мјера тога колико политика доводи до опажања која су у складу са преференцијама агента.

Оба ова члана се сабирају за сваки корак унутар политике, а њихов збир представља укупну очекивану слободну енергију политике. Нижа вриједност G означава пожељнију политику — ону која води ка предвидљивијим (мање неодређеним) и пожељнијим опажањима,

```
def calculate_G_policies(A, B, C, qs_current, policies):

  G = np.zeros(len(policies)) # initialize the vector of expected free
   energies, one per policy
  H_A = entropy(A)            # can calculate the entropy of the A
   matrix beforehand, since it'll be the same for all policies

  for policy_id, policy in enumerate(policies): # loop over policies -
   policy_id will be the linear index of the policy (0, 1, 2, ...) and
   `policy` will be a column vector where `policy[t,0]` indexes the
```

```
    action entailed by that policy at time ‘t‘

    t_horizon = policy.shape[0] # temporal depth of the policy

    G_pi = 0.0 # initialize expected free energy for this policy

    for t in range(t_horizon): # loop over temporal depth of the policy

      action = policy[t,0] # action entailed by this particular policy,
    at time ‘t‘

      # get the past predictive posterior - which is either your
    current posterior at the current time (not the policy time) or the
    predictive posterior entailed by this policy, one timstep ago (in
    policy time)
      if t == 0:
        qs_prev = qs_current
      else:
        qs_prev = qs_pi_t

      qs_pi_t = get_expected_states(B, qs_prev, action) # expected
    states, under the action entailed by the policy at this particular
    time
      qo_pi_t = get_expected_observations(A, qs_pi_t)   # expected
    observations, under the action entailed by the policy at this
    particular time

      kld = kl_divergence(qo_pi_t, C) # Kullback-Leibler divergence
    between expected observations and the prior preferences C

      G_pi_t = H_A.dot(qs_pi_t) + kld # predicted uncertainty +
    predicted divergence, for this policy & timepoint

      G_pi += G_pi_t # accumulate the expected free energy for each
    timepoint into the overall EFE for the policy

    G[policy_id] += G_pi

  return G
```

Код 2.1.13: Функција calculate_G_policies

У наставку је дата функција која израчунава вјероватноће избора сваке појединачне акције на основу вјероватноћа додељених свакој политици (Q_π). Пошто се одлука о акцији доноси у садашњем тренутку, узима се у обзир само прва акција

у свакој политици. Умјесто сабирања, користи се максимална вриједност Q_π за сваку акцију (односно највећа вјероватноћа неке политике која ту акцију предвиђа као прву), што значи да се као изабрана узима акција са почетка политике која има највећу вјероватноћу. На крају, резултујући вектор се нормализује како би представљао валидну дистрибуцију вероватноћа над акцијама.

```
def compute_prob_actions(actions, policies, Q_pi):
  P_u = np.zeros(len(actions)) # initialize the vector of probabilities
    of each action

  for policy_id, policy in enumerate(policies):
    P_u[int(policy[0,0])] = max(P_u[int(policy[0,0])], Q_pi[policy_id])

  P_u = utils.norm_dist(P_u) # normalize the action probabilities

  return P_u
```

Код 2.1.14: Функција compute_prob_actions

Функција active_inference_with_planning представља главну петљу агента који користи приступ Active Inference са планирањем да би доносио одлуке и интераговао са окружењем током задатог броја временских корака. На почетку, агент поставља своје почетно увјерење о скривеном стању користећи матрицу D, ресетује окружење и добија прво опажање. Такође конструише све могуће политике – низове акција одређене дужине – које ће користити за планирање.

У сваком кораку, агент најпре опсервира своје тренутно стање у окружењу. Ако је достигао циљну локацију (тј. координату 47), прекида епизоду и враћа коначно увјерење и акумулирану награду. У супротном, врши инференцију над скривеним стањем на основу запажања и матрице A, користећи Бајесово правило. Ово увјерење представља његово тренутно разумијевање гдје се налази у простору скривених стања.

На основу овог увјерења, агент симулира сваки могући низ акција (свака политика) и израчунава њихову очекивану слободну енергију, користећи функцију calculate_G_policies. Ова вриједност укључује предвиђену неизвјесност у опажањима као и разлику између предвиђених и пожељних опажања. Затим, примјењује softmax функцију над негативним вредностима слободне енергије да би добио вероватноће избора сваке политике (Q_π), и на основу тога израчунава маргиналну вероватноћу избора сваке акције (P_u). Из P_u се затим бира акција са највећом вероватноћом.

Након избора акције, агент предвиђа расподјелу скривених стања која би могла настати као последица те акције, користећи транзициону матрицу B и своје текуће увјерење. Креће се у окружењу, добија ново опажање и награду, и наставља циклус све док не истекне вријеме, или се епизода не заврши услед постизања циља. На крају, функција враћа коначно ујверење о скривеном стању и укупну награду прикупљену током епизоде.

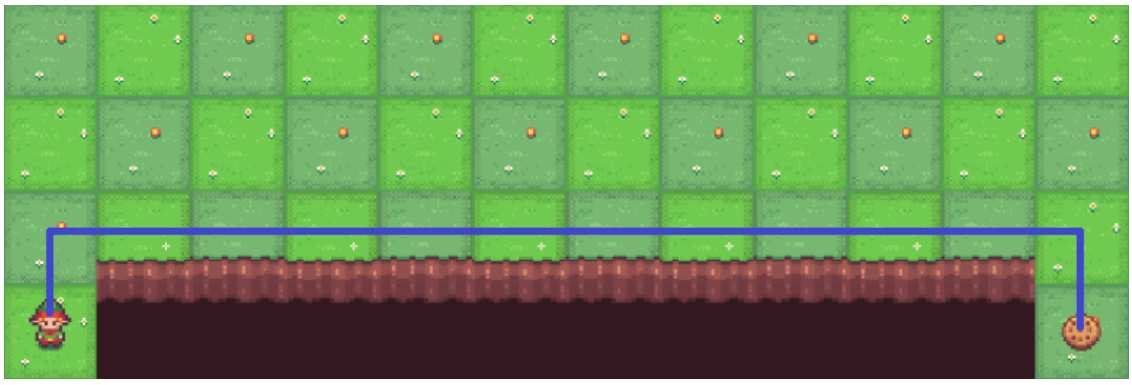
```
def active_inference_with_planning(A, B, C, D, n_actions, env,
    policy_len=2, T=5):

    """ Initialize prior, first observation, and policies """
    prior = D
    obs, _ = env.reset()
    cumulative_reward = 0

    policies = construct_policies([n_states], [n_actions], policy_len=
    policy_len)


    for t in range(T):
        print(f'\nTime {t}: Agent observes itself in location: {obs} (
    Grid: {grid_locations[obs]})')

        if obs == 47:
            print(f"Agent reached the target at {grid_locations[obs]}!
    Stopping.")
            return qs_current, cumulative_reward  # Stop and return

        # Inference (use the integer observation directly)
        qs_current = infer_states(obs, A, prior)

        # Calculate expected free energy
        G = calculate_G_policies(A, B, C, qs_current, policies)
        Q_pi = softmax(-G)
        P_u = compute_prob_actions(actions, policies, Q_pi)
        chosen_action = np.argmax(P_u)

        # Update prior
        prior = B[:, :, chosen_action].dot(qs_current)

        next_obs, reward, terminated, truncated, info = env.step(
    chosen_action)

        cumulative_reward += reward
```

```
33
34
35          print(f"Chosen action: {actions[chosen_action]} | Reward: {
    reward}")
36          frame = env.render()  # returns an RGB array since env was
    created with render_mode="rgb_array"
37          display_frame(frame, title_str=f"Time {t + 1}: Action {actions[
    chosen_action]}")
38
39          if terminated or truncated:
40              print(f"Episode terminated! Total reward: {
    cumulative_reward}")
41              return qs_current, cumulative_reward
42          obs = next_obs
43
44      print(f"\nFinal cumulative reward after {T} timesteps: {
    cumulative_reward}")
45
46      return qs_current, cumulative_reward
```

Код 2.1.15: Функција active_inference_with_planning

2.1.5 Понашање агента

Након извршене имплементације, посматрајмо понашање агента у симулираном окружењу. Циљ агента је да стигне до задате локације, при чему се очекује да прелази што краћи пут и избјегава провалије. На доњој слици приказана је оптимална путања од почетне позиције до циљне локације. За тај пут му је потребно да направи 13 корака.



Слика 2.4: Оптимална путања агента

У оквиру тестирања, агент разматра политике дужине 7 корака унапријед (policy_len = 7), док је максимално трајање епизоде ограничено на 20 временских корака (T = 20).

Резултати симулације показују да агент, вођен принципом Active Inference, доследно прати ову руту. Ово указује да агент успјешно планира низ акција које минимизују очекивану слободну енергију, чиме постиже и функционално и рационално понашање у датом окружењу.

2.2 Модел са шумом у матрицама A и D

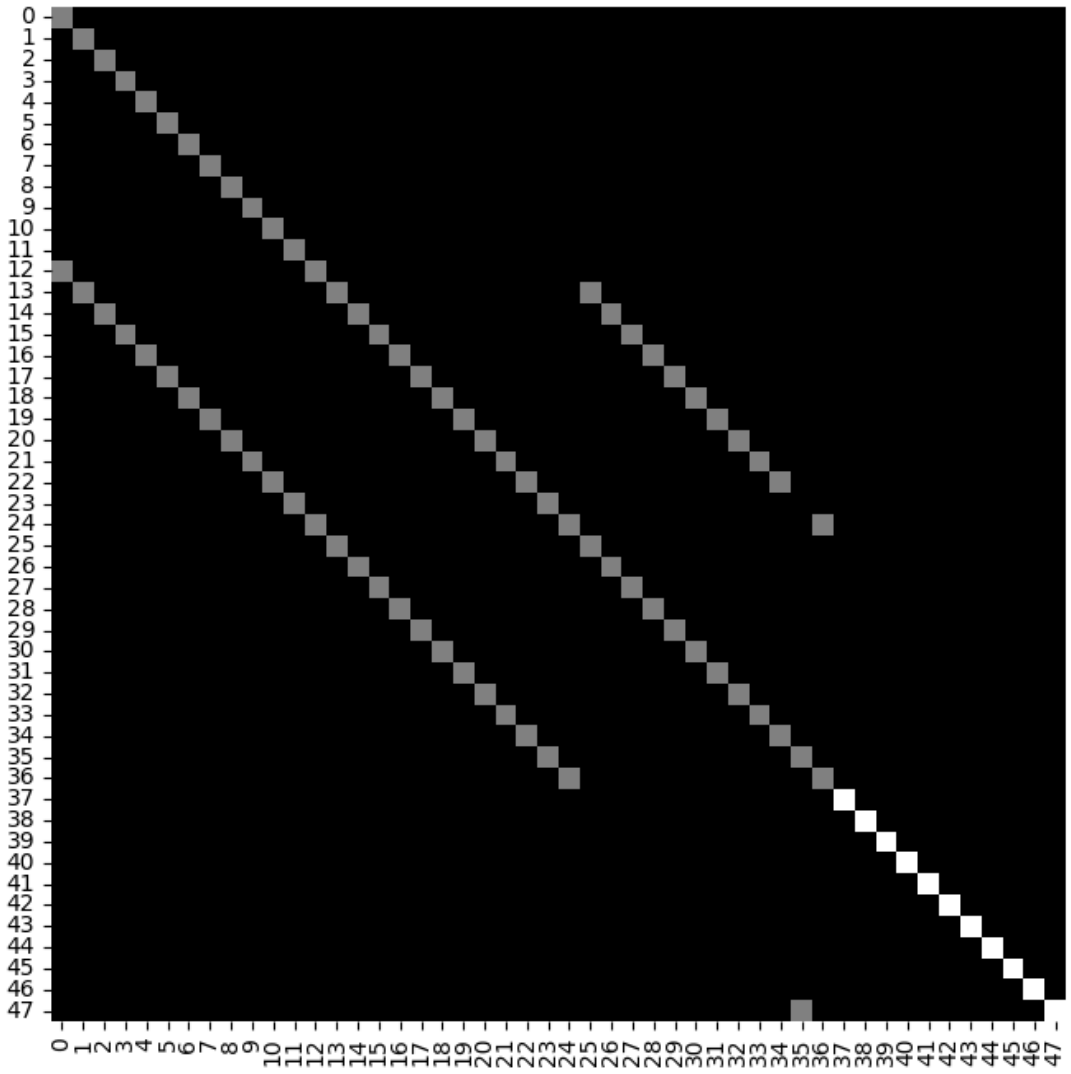
У овој варијанти модела, уводи се додатна неодређеност у опажањима и почетно увјерење агента, како би се симулирали услови ограничене или зашумљене перцепције. У циљу моделирања ситуације у којој агент нема потпуно поуздана сензорна опажања и у којој његова почетна увјерења о локацији нису прецизна, модификован је основни модел у неколико кључних аспеката.

Најприје, матрица опажања A проширена је тако да не одражава више детерминистичку везу између стања и опажања. Конкретно, ако опсервира да је у нултом реду, постоји једнака вјероватноћа да је у опаженом или у стању испод. Слично томе, када агент опази да се налази у стању у првом или другом реду мреже, модел му даје једнаку вјероватноћу да се заправо налази у опаженом стању или у оном изнад. Посебан случај је стање 36 (почетно стање), гдје се такође укључује неизвјесност према горњем стању. Ове вјероватноће симулиране су тако што су одговарајуће вриједности у матрици A постављене на једнаке бројеве, а затим је матрица нормализована по колонама. На тај начин, опажања агента више нису поуздан одраз стварног стања, већ садрже шум.

```
A = np.eye(n_observations, n_states)

for i in range(n_observations):
  if 0 <= i <= 11:
    A[i+12, i] = 1
  elif 12 <= i <= 35:
    if 24 < i < 35:
      A[i-12, i] = 1
    else:
      A[i+12, i] = 1
  elif i == 36:
      A[i-12, i] = 1

```

```
14 A = A / A.sum(axis=0, keepdims=True)
15
16
17 plot_likelihood(A)
```

Код 2.2.1: Модификована матрица A

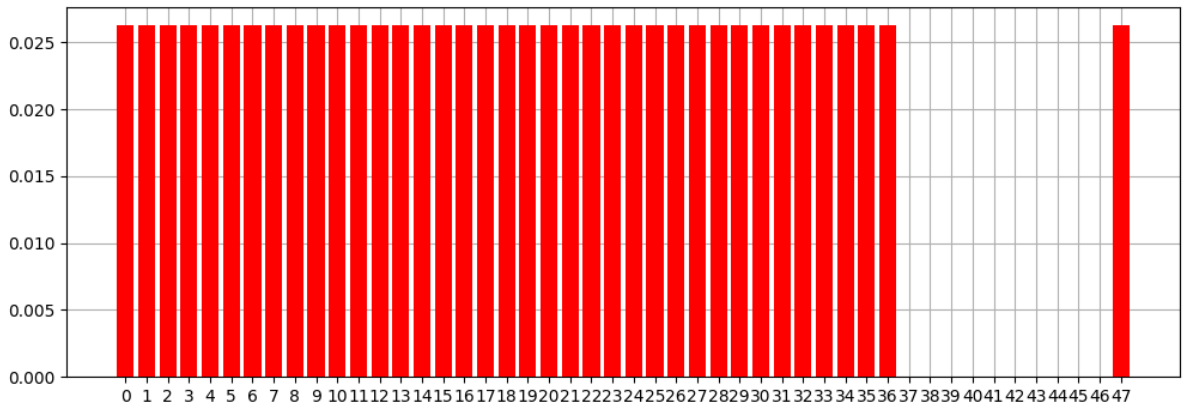


Слика 2.5: График вјероватноће запажања у зависности од скривених стања са шумом

Поред тога, измјењена је и почетна расподјела увјерења, односно матрица D. Вриједности у овој дистрибуцији су подешене тако да стања која представљају литица (индекси 37–46) имају вјероватноћу једнаку нули, након чега је цијела расподјела нормализована. На овај начин агент на самом почетку нема никакву претпоставку о томе у ком се конкретном стању налази, већ креће са равномјерном неизвјесношћу над преосталим стањима. Оваква иницијализација подстиче агента да кроз кретање и опажања постепено изгради слику о својој позицији у окружењу.

```
D = np.ones(n_states)

D[37:47] = 0

D = D / D.sum()

plot_beliefs(D)
```

Код 2.2.2: Модификована матрица D



Слика 2.6: График вјероватноће почетних стања са равномјерном расподјелом над дозвољеним позицијама

Да би се агент подстакао на истраживање окружења у условима повећане неизвјесности, извршене су двије кључне измјене у моделу.

У функцији compute_prob_actions, начин обраде вјероватноћа појединачних акција промијењен је са избора политике са највећом вјероватноћом (max) на сабирање доприноса свих политика (+ =). Ова промјена значи да ће се вјероватноћа одређене

акције повећавати пропорционално томе колико се често та акција појављује као прва у високовјероватним политикама.

```
def compute_prob_actions(actions, policies, Q_pi):
  P_u = np.zeros(len(actions)) # initialize the vector of probabilities
    of each action

  for policy_id, policy in enumerate(policies):
    P_u[int(policy[0,0])] += Q_pi[policy_id] # get the marginal
   probability for the given action, entailed by this policy at the
   first timestep

  P_u = utils.norm_dist(P_u) # normalize the action probabilities

  return P_u
```

Код 2.2.3: Измијењена compute_prob_actions функција

На крају, избор акције више није базиран на детерминистичком критеријуму (argmax), већ се користи стохастички избор преко utils_sample. Овај приступ омогућава да се и акције са нешто мањом вјероватноћом повремено бирају, што директно подржава истраживање. Тако се боље истиче значај направљених промјена у условима неизвјесности агента о његовом тренутном положају и окружењу.

```
def active_inference_with_planning(...):
    # dio funkcije prije izbora akcije...

    # Izbor akcije: ranije je birana akcija sa najvecom vjerovatnocom

    # chosen_action = np.argmax(P_u)

    # Sada se koristi uzorkovanje iz raspodjele, da bi se podstaklo
   istrazivanje:
    chosen_action = utils.sample(P_u)

    # ostatak funkcije...
```

Код 2.2.4: active_inference_with_planning

Током тестирања коришћена је дужина политике (policy_len) једнака 7 и дужина епизоде (T) је постављена на 30 корака.

Агент се не понаша увијек најоптималније, јер због имплементираног истраживања понекад бира акције које нису најкраћи пут до циља. Међутим, и поред тога, он

увијек успијева да стигне до циља у релативно малом броју корака. Такође, агент вјешто избјегава литице и никада не упада у њих. Иако често није сигуран да ли се налази у опсервираном пољу или у пољу изнад њега, оптималне путање у оба случаја се у великој мјери поклапају, па ова неизвјесност не доводи до озбиљних проблема у кретању. У неким другим сценаријима, са другачије дефинисаним матрицама A и D, ова врста перцептивне неодређености би могла више утицати на понашање агента.

2.3 Модел са клизавом површином (is_slippery=True)

У овој варијанти модела уводи се додатна неодређеност у кретању агента кроз окружење, с циљем симулирања услова кретања по клизавој подлози. Агент се са вјероватноћом од $1/3$ креће у намјераваном правцу, док постоји једнака вјероватноћа од по $1/3$ да ће завршити у једном од два правца која су нормална у односу на онај који је изабрао. Ова промјена уводи неизвјесност у саму динамику окружења, захтијевајући од агента да прилагоди процес планирања и доношења одлука у условима у којима више нема потпуну контролу над сопственим кретањем.

У односу на стандардни модел, једина промјена у овој варијанти односи се на дефинисање транзиционе матрице B, која описује вјероватноће преласка из једног скривеног стања у друго као резултат одређене акције. Док је у основном моделу прелаз био детерминистички, овдје је уведена стохастичка компонента која симулира клизаву подлогу: агент се са вјероватноћом од $1/3$ креће у намјераваном правцу, а са по $1/3$ у један од два правца која су нормална на жељени.

На тај начин, матрица B је прилагођена тако да, у зависности од тренутне позиције и намјераване акције, расподјели вјероватноће на три могуће наредне позиције. Испод је приказан код којим је дефинисана нова транзициона матрица:

```
def create_B_matrix():
  B = np.zeros( (len(grid_locations), len(grid_locations), len(actions)
   ) )

  for action_id, action_label in enumerate(actions):

    for curr_state, grid_location in enumerate(grid_locations):

      y, x = grid_location

      if (y == 3 and 1 <= x <= 10):
        x = 0
        y = 3
```

```
13
14      if action_label == "UP":
15        next_y = y - 1 if y > 0 else y
16        next_x = x
17        next_x1 = max(x - 1, 0)
18        next_y1 = y
19        next_x2 = min(x + 1, 11)
20        next_y2 = y
21      elif action_label == "DOWN":
22        next_y = y + 1 if y < 3 else y
23        next_x = x
24        next_x1 = max(x - 1, 0)
25        next_y1 = y
26        next_x2 = min(x + 1, 11)
27        next_y2 = y
28      elif action_label == "LEFT":
29        next_x = x - 1 if x > 0 else x
30        next_y = y
31        next_x1 = x
32        next_y1 = max(y - 1, 0)
33        next_x2 = x
34        next_y2 = min(y + 1, 3)
35      elif action_label == "RIGHT":
36        next_x = x + 1 if x < 11 else x
37        next_y = y
38        next_x1 = x
39        next_y1 = max(y - 1, 0)
40        next_x2 = x
41        next_y2 = min(y + 1, 3)
42
43
44      new_location = (next_y, next_x)
45      new_location1 = (next_y1, next_x1)
46      new_location2 = (next_y2, next_x2)
47      next_state = grid_locations.index(new_location)
48      next_state1 = grid_locations.index(new_location1)
49      next_state2 = grid_locations.index(new_location2)
50
51      if next_state != next_state1 and next_state != next_state2:
52        B[next_state, curr_state, action_id] = 1/3
53        B[next_state1, curr_state, action_id] = 1/3
54        B[next_state2, curr_state, action_id] = 1/3
55      elif next_state == next_state1 and next_state != next_state2:
56        B[next_state, curr_state, action_id] = 2/3
57        B[next_state2, curr_state, action_id] = 1/3
```

```
58        elif next_state != next_state1 and next_state == next_state2:
59          B[next_state, curr_state, action_id] = 2/3
60          B[next_state1, curr_state, action_id] = 1/3
61        else:
62          B[next_state, curr_state, action_id] = 1.0
63
64
65    return B
66
67  B = create_B_matrix()
```

Код 2.3.1: Дефинисање матрице B у моделу са клизавом подлогом

Током тестирања коришћена је дужина политике policy_len = 2, што се показало као погодан избор, будући да агент не може прецизно да предвиди позицију у даљој будућности, па краће политике боље одражавају значај актуелних одлука. Максимална дужина епизоде постављена је на T = 60.

Подсјетимо се овдје изгледа окружења. Када се агент налази у почетном стању (стање 36), он бира акцију LEFT. Оваква одлука минимизира ризик од пада у литицу (која се налази десно), а омогућава могућност да се агент омакне у горњи ред, док у најгорем случају остаје у истом положају. Након што се позиционира у реду 2, бира акцију UP, чиме избјегава опасну зону испод, а има могућност да се помјери и бочно. Када дође у ред 1, почиње да се креће RIGHT кроз мрежу, настављајући тако и у случају да се омакне у ред 0. Овај образац кретања – вертикално уздизање ка безбједнијој зони, а затим хоризонтално кретање ка циљу – омогућава агенту да избјегне литице и да постепено напредује.

По доласку у последњу колону, агент испитује своју позицију. Ако се налази у стању 11 или 23 (нулти и први ред), користи акцију DOWN да би дошао до стања 35. Уколико је већ у стању 35, бира акцију RIGHT, ослањајући се на могућност да се омакне у циљно стање 47 које се налази испод.

Због клизаве површине и сталне могућности непредвиђеног помјерања, агент веома тешко стиже до циља — али његово понашање и даље одражава оптималну адаптацију у датом окружењу.