

گزارش پروژه پایانی درس مبانی و کاربردهای هوش مصنوعی

استاد: سرکار خانم دکتر طباطبایی

استادیار: جناب آقای زادضیابری

پدیدآورندگان: مهدی شیرازی، آریا باقرزاده فر

بهار ۱۴۰۴

مقدمه

این گزارش به تحلیل و بررسی پروژه روبات کاوشگر مریخ اَستروبات (AstroBot) می‌پردازد، که با استفاده از دانش فنی و الگوریتمی کسب شده در طول ترم، کارکردهای مختلف این مریخ‌نورد، مانند مسیریابی و تقسیم وظایف را با استفاده از زبان پایتون طراحی و پیاده‌سازی کردیم.

فایل پروژه حاوی ۳ بخش است:

1. پوشه codes: این پوشه حاوی فایل‌های مرتبط با هر ۳ قسمت پروژه، اعم از کد به زبان پایتون و فایل‌های Jupyter Notebook است.
2. فایل .docx (این فایل): گزارشی از نحوه پیاده‌سازی پروژه
3. فایل readme.md: ابزارهای بکاررفته در پیاده‌سازی این پروژه در این فایل ذکر شده است

در ادامه به بررسی پیاده‌سازی بخش‌های مختلف پروژه می‌پردازیم.

بخش اول: مسیریابی

در این بخش از پروژه به بهینه‌سازی سیستم مسیریابی مریخ‌نورد را بررسی می‌کنیم.

در این سناریو، روبات نیاز دارد که در یک نقشه ۵×۵ بهینه‌ترین مسیر را از نقطه (۰,۰) به منبع آب در نقطه (۴,۴) پیدا کند.

داده‌های مسئله:

1. یک شبکه ۵×۵ سلولی که محیط فرود کاوشگر را شبیه‌سازی می‌کند
2. سلول‌های با مقدار ۰ که نشانگر مناطق قابل عبور هستند (عبور از این سلول‌ها مجاز است)
3. سلول‌های با مقدار ۱ که نشانگر صخره‌ها اند (عبور از این سلول‌ها ممنوع است)
4. حرکت در چهار جهت بالا، پایین، چپ و راست (حرکت مورب ممنوع است)
5. نقطه شروع: (۰,۰)
6. نقطه پایان: (۴,۴)

در این مسئله، با توجه به قابلیت‌های الگوریتم BFS (Breadth-First Search) در پیدا کردن بهینه‌ترین مسیر در محیط‌های کوچک و سادگی آن، به سراغ این الگوریتم رفتیم. مراحل الگوریتم پیاده‌سازی شده به شرح زیر است:

1. شروع از نقطه اولیه (۰,۰) و افزودن به صف
2. بررسی همسایه‌های چهارگانه سلول جاری
3. افزودن سلول‌های معتبر به صف
4. تکرار فرآیند تا رسیدن به نقطه هدف
5. ثبت مسیر نهایی

ساختارهای داده

ماتریس ۵×۵ برای ذخیره نقشه:

grid: list[list[int]] = [

[0, 0, 0, 1, 0],

[1, 1, 0, 1, 0],

[0, 0, 0, 1, 0],

[0, 1, 1, 1, 0],

[0, 0, 0, 0, 0]

]

صف (Queue) برای مدیریت سلول‌های در حال بررسی:

from collections import deque

queue = deque()

مجموعه (Set) برای ذخیره سلول‌های بازدید شده:

visited = set()

توابع:

تابع بررسی صحت نقشه (gridValidation):

این تابع به بررسی ابعاد ماتریس داده شده (به صورت list) می‌پردازد و اگر مرتبه ماتریس ۵ نبود یا مقادیر سلول‌های آن غیر از صفر و یک بودند، نتیجه آن عدم صحت نقشه خواهد بود.

def gridValidation(grid) -> bool:

if len(grid) != 5 or any(len(row) != 5 for row in grid):

return False

for row in grid:

for cell in row:

if cell not in {0, 1}:

return False

return True

تابع پیاده‌سازی الگوریتم BFS:

def BFS\_Algorithm(grid) -> list or str:

if not gridValidation(grid):

return "Invalid input. Grid must be 5x5 with only 0s and 1s."

if grid[0][0] == 1 or grid[4][4] == 1:

return "No path found. Start or end position is blocked."

directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

queue = deque()

queue.append((0, 0, [(0, 0)]))

visited = set()

visited.add((0, 0))

while queue:

row, col, path = queue.popleft()

if row == 4 and col == 4:

return path

for dr, dc in directions:

new\_row, new\_col = row + dr, col + dc

if 0 <= new\_row < 5 and 0 <= new\_col < 5:

if grid[new\_row][new\_col] == 0 and (new\_row, new\_col) not in visited:

visited.add((new\_row, new\_col))

queue.append((new\_row, new\_col, path + [(new\_row, new\_col)]))

return "No path exists to the destination."

نتایج و تحلیل:

حال عملکرد الگوریتم را در ۲ سناریو بررسی می‌کنیم:

مسیر آسان:

ورودی:

grid = [

[0, 0, 0, 0, 0],

[0, 1, 1, 1, 0],

[0, 1, 0, 1, 0],

[0, 1, 0, 1, 0],

[0, 0, 0, 0, 0]

]

خروجی:

[(0,0), (0,1), (0,2), (0,3), (0,4), (1,4), (2,4), (3,4), (4,4)]

مسیر سخت

ورودی:

grid = [

[0, 0, 0, 0, 0],

[0, 1, 1, 1, 0],

[0, 1, 0, 1, 0],

[0, 1, 0, 1, 0],

[0, 0, 0, 0, 0]

]

خروجی:

[(0,0), (1,0), (2,0), (2,1), (2,2), (1,2), (0,2), (0,3), (0,4), (1,4), (2,4), (3,4), (4,4)]

تحلیل عملکرد:

حال عملکرد این الگوریتم را بررسی می‌کنیم:

|  |  |
| --- | --- |
| معیار | عملکرد |
| زمان اجرا | کمتر از ۰.۰۱ ثانیه |
| فضای اشغال شده | کمتر از ۱ مگابایت |
| بیشترین تعداد مراحل | ۲۵ |

الگوریتم توسعه داده شده نشان‌دهنده یک راه‌حل پایدار و کارآمد برای مسئله مسیریابی کاوشگرهای مریخی در محیط‌های محدود است. الگوریتم BFS با وجود سادگی، عملکرد مناسبی در یافتن کوتاه‌ترین مسیر دارد.

بخش دوم: جانمایی پنل‌های خورشیدی

در این بخش از پروژه، رویکرد مریخ‌نورد در جانمایی پنل‌های خورشیدی را بررسی می‌کنیم. برای حل این مسئله و پیاده‌سازی آن از الگوریتم ژنتیک بهره بردیم که شرح حل مسئله به کمک آن به صورت زیر می‌باشد:

1. نمایش (Representation): هر راه‌حل (ژنوم) به صورت لیستی از ۱۰ بیت (0 و 1) نمایش داده می‌شود که دقیقاً ۵ بیت از این ۱۰ بیت مقدار ۱ دارند؛ یعنی انتخاب ۵ جایگاه به عنوان محل نصب پنل‌های خورشیدی
2. تابع تناسب (Fitness Function): تناسب یک ژنوم برابر است با مجموع مقادیر نوردهی (exposures) در مکان‌هایی که ژنوم دارای بیت ۱ است. این تابع کیفیت یک راه‌حل را نشان می‌دهد؛ هرچه مجموع نوردهی بیشتر باشد، آن ژنوم بهتر است.

بخش سوم: دسته‌بندی و محول کردن وظایف

در این بخش از پروژه، به طراحی و پیاده‌سازی یک سیستم هوشمند زمان‌بندی وظایف برای مریخ‌نورد می‌پردازیم. با توجه به محدودیت‌های سخت‌افزاری و عملیاتی در محیط مریخ، نیاز به یک سیستم برنامه‌ریزی دقیق برای مدیریت بهینه منابع انرژی و زیرسیستم‌ها احساس می‌شود.

در این بخش از پروژه، با چند دسته چالش مواجه هستیم:

1. محدودیت شدید منابع انرژی
2. چالش‌های محاسباتی و الگوریتمی
3. چالش‌های عملیاتی
4. چالش‌های پیاده‌سازی

حال با توجه به شرایط مسئله، باید الگوریتمی طراحی کنیم که ضمن رعایت دقیق محدودیت‌های انرژی و جلوگیری از تداخل عملکردی زیرسیستم‌ها، ۵ وظیفه اصلی را در ۵ بازه زمانی به صورت بهینه تقسیم بندی و توزیع کرده و تضمین کند که تمام وظایف در بازه‌های مشخص انجام می‌شوند.

ساختارهای داده و ورودی‌ها:

ساختار داده Dictionary برای ذخیره وظایف و اطلاعات مربوطه به صورت زوج‌هایی از keyها و valueها:

subsystems: Dict[str, str] = {

'T1': 'Navigation',

'T2': 'Sampling',

'T3': 'Communication',

'T4': 'Navigation',

'T5': 'Sampling'

}

power\_needs: Dict[str, int] = {

'T1': 5,

'T2': 4,

'T3': 6,

'T4': 7,

'T5': 3

}

ساختار داده Tuple برای ذخیره داده‌های ثابت که نیازی به تغییر ندارند:

tasks: tuple = ('T1', 'T2', 'T3', 'T4', 'T5')

power\_limits: tuple = (10, 6, 12, 8, 10)

توابع:

تابع backtrack که هسته اصلی الگوریتم است. این تابع هدفش این است که هر وظیفه را به یکی از بازه‌های زمانی ممکن به شیوه ای اختصاص دهد که محدودیت‌های زیرسیستم‌ها (subsystems) و محدودیت‌های انرژی مطابقت داشته باشد. اگر تخصیص معتبری پیدا کند، آن را برمی‌گرداند؛ در غیر این صورت، به عقب برمی‌گردد تا ترکیب دیگری را امتحان کند.

def backtrack(assignment, tasks, subsystems, power\_needs, power\_limits):

if len(assignment) == len(tasks):

return assignment

for task in tasks:

if task not in assignment:

for time\_slot in range(1, 6):

if isValidAssignment(assignment, task, time\_slot, subsystems, power\_needs, power\_limits):

assignment[task] = time\_slot

result = backtrack(assignment.copy(), tasks, subsystems, power\_needs, power\_limits)

if result is not None:

return result

break

return None

تابع isValidAssignment که وظیفه بررسی صحت شیوه محول شدن وظایف را بر عهده دارد:

def isValidAssignment(assignment, task, time\_slot, subsystems, power\_needs, power\_limits):

if (time\_slot in assignment.values()) or power\_needs[task] > power\_limits[time\_slot - 1]:

return False

current\_subsystem = subsystems[task]

for adj\_time in [time\_slot - 1, time\_slot + 1]:

if adj\_time in assignment.values():

for t, slot in assignment.items():

if slot == adj\_time and subsystems[t] == current\_subsystem:

return False

return True

تحلیل عملکرد الگوریتم:

مکانیزم عملکرد الگوریتم پیاده‌سازی شده به شرح زیر است:

1. انتخاب وظیفه: سیستم به صورت متوالی هر وظیفه را انتخاب می‌کند
2. تخصیص زمانی: بررسی تمام بازه‌های ۱ تا ۵ برای هر وظیفه
3. بررسی محدودیت‌ها: سه فیلتر اصلی اعمال می‌شود
4. تکرار بازگشتی: فرآیند برای وظایف باقیمانده تکرار می‌شود
5. عقبگرد: اگر مسیری به جواب نرسد، به مرحله قبل برمی‌گردد

چالش‌های محاسباتی و الگوریتمی:

در فرایند توسعه، با چالش‌های زیر مواجه شدیم:

1. انفجار ترکیبی (Combinatorial Explosion): با افزایش تعداد وظایف (n)، پیچیدگی محاسباتی به صورت فاکتوریل (O(n!)) رشد می‌کند. برای مثال، برای ۵ وظیفه ۱۲۰ حالت ممکن و برای ۱۰ وظیفه، ۳.۶ میلیون حالت ممکن وجود دارد.
2. بهینه‌سازی مصرف حافظه: نیاز به ذخیره‌سازی تمام تخصیص‌های موقت در حین بازگشت
3. چالش‌های عملیاتی: از این دسته می‌توان به تداخل زیرسیستم‌ها (برای مثال، سیستم ناوبری هنگام نمونه ‌برداری دقت کمتری دارد) و نیاز به افزودن محدودیت‌های غیرخطی (برای مثال، اگر T4 فعال باشد، T1 حداکثر ۳ واحد انرژی می‌تواند مصرف کند) اشاره کرد.

راهکارهای پیشنهادی:

1. استفاده از هیوریستیک‌ها: اولویت‌دهی به وظایف با انرژی کمتر در بازه‌های محدود
2. پیاده‌سازی نسخه موازی: تقسیم مسئله بین چند پردازنده
3. بهره‌گیری از یادگیری ماشین: پیش‌بینی مصرف انرژی واقعی بر اساس داده‌های تاریخی

محدودیت ها:

در این قسمت، به برخی از محدودیت‌های الگوریتم پیاده‌سازی شده اشاره شده است:

1. برای n>8 غیر عملی می شود.
2. پیچیدگی زمانی بسیار بالا (O(n2 n))
3. پیچیدگی فضایی نسبتاً مناسب (O(n))

ضمیمه

کد بخش اول پروژه:

from collections import deque

def gridValidation(grid) -> bool:

if len(grid) != 5 or any(len(row) != 5 for row in grid):

return False

for row in grid:

for cell in row:

if cell not in {0, 1}:

return False

return True

def BFS\_Algorithm(grid) -> list or str:

if not gridValidation(grid):

return "Invalid input. Grid must be 5x5 with only 0s and 1s."

if grid[0][0] == 1 or grid[4][4] == 1:

return "No path found. Start or end position is blocked."

directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

queue = deque()

queue.append((0, 0, [(0, 0)]))

visited = set()

visited.add((0, 0))

while queue:

row, col, path = queue.popleft()

if row == 4 and col == 4:

return path

for dr, dc in directions:

new\_row, new\_col = row + dr, col + dc

if 0 <= new\_row < 5 and 0 <= new\_col < 5:

if grid[new\_row][new\_col] == 0 and (new\_row, new\_col) not in visited:

visited.add((new\_row, new\_col))

queue.append((new\_row, new\_col, path + [(new\_row, new\_col)]))

return "No path exists to the destination."

def main() -> None:

grid: list[list[int]] = [

[0, 0, 0, 1, 0],

[1, 1, 0, 1, 0],

[0, 0, 0, 1, 0],

[0, 1, 1, 1, 0],

[0, 0, 0, 0, 0]

]

path = BFS\_Algorithm(grid)

print("Path found:", path)

if \_\_name\_\_ == "\_\_main\_\_":

main()

کد بخش دوم پروژه:

کد بخش سوم پروژه:

def isValidAssignment(assignment, task, time\_slot, subsystems, power\_needs, power\_limits):

if time\_slot in assignment.values():

return False

if power\_needs[task] > power\_limits[time\_slot - 1]:

return False

current\_subsystem = subsystems[task]

for adj\_time in [time\_slot - 1, time\_slot + 1]:

if adj\_time in assignment.values():

for t, slot in assignment.items():

if slot == adj\_time and subsystems[t] == current\_subsystem:

return False

return True

def backtrack(assignment, tasks, subsystems, power\_needs, power\_limits):

if len(assignment) == len(tasks):

return assignment

for task in tasks:

if task not in assignment:

for time\_slot in range(1, 6):

if isValidAssignment(assignment, task, time\_slot, subsystems, power\_needs, power\_limits):

assignment[task] = time\_slot

result = backtrack(assignment.copy(), tasks, subsystems, power\_needs, power\_limits)

if result is not None:

return result

break

return None

def main() -> None:

tasks: tuple = ('T1', 'T2', 'T3', 'T4', 'T5')

subsystems: Dict[str, str] = {

'T1': 'Navigation',

'T2': 'Sampling',

'T3': 'Communication',

'T4': 'Navigation',

'T5': 'Sampling'

}

power\_needs: Dict[str, int] = {

'T1': 5,

'T2': 4,

'T3': 6,

'T4': 7,

'T5': 3

}

power\_limits: tuple = (10, 6, 12, 8, 10)

assignment = backtrack({}, tasks, subsystems, power\_needs, power\_limits)

if assignment:

print("Task assignment found:")

for task, time\_slot in assignment.items():

print(f"{task}: Time slot {time\_slot} (Subsystem: {subsystems[task]}, Power: {power\_needs[task]})")

else:

print("No solution found")

if \_\_name\_\_ == "\_\_main\_\_":

main()

منابع

<https://github.com/omonimus1/geeks-for-geeks-solutions/blob/master/c++/bfs-graph.cpp>

<https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>

<https://en.wikipedia.org/wiki/Backtracking>

<https://www.geeksforgeeks.org/backtracking-algorithms/>

<https://www.youtube.com/watch?v=DKCbsiDBN6c>

[ChatGPT](https://chatgpt.com/)

[Grok](https://grok.com/)