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:

۱. پوشه codes: این پوشه حاوی فایل‌های مرتبط با هر ۳ قسمت پروژه، اعم از کد به زبان پایتون و فایل‌های Jupyter Notebook است.
۲. فایل docx. (این فایل): گزارشی از نحوه پیاده‌سازی پروژه
۳. 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:

۱. یک شبکه ۵×۵ سلولی که محیط فرود کاوشگر را شبیه‌سازی می‌کند
۲. سلول‌های با مقدار ۰ که نشانگر مناطق قابل عبور هستند (عبور از این سلول‌ها مجاز است)
۳. سلول‌های با مقدار ۱ که نشانگر صخره‌ها اند (عبور از این سلول‌ها ممنوع است)
۴. حرکت در چهار جهت بالا، پایین، چپ و راست (حرکت مورب ممنوع است)
۵. نقطه شروع: (۰,۰)
۶. 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:

۱. شروع از نقطه اولیه (۰,۰) و افزودن به صف
۲. بررسی همسایه‌های چهارگانه سلول جاری
۳. افزودن سلول‌های معتبر به صف
۴. تکرار فرآیند تا رسیدن به نقطه هدف
۵. 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 به بهینه‌سازی سیستم مسیریابی مریخ‌نورد را بررسی می‌کنیم.

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:

۱. محدودیت شدید منابع انرژی
۲. چالش‌های محاسباتی و الگوریتمی
۳. چالش‌های عملیاتی
۴. 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:

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

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

Grok