



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

این گزارش به تحلیل و بررسی پروژه روبات کاوشگر مریخ آستروبات (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 5×5 بهینه‌ترین مسیر را از نقطه $(0,0)$ به منبع آب در نقطه $(4,4)$ پیدا کند.

داده‌های مسئله:

- یک شبکه 5×5 سلولی که محیط فرود کاوشگر را شبیه‌سازی می‌کند
 - سلول‌های با مقدار ۰ که نشانگر مناطق قابل عبور هستند (عبور از این سلول‌ها مجاز است)
 - سلول‌های با مقدار ۱ که نشانگر صخره‌ها اند (عبور از این سلول‌ها ممنوع است)
 - حرکت در چهار جهت بالا، پایین، چپ و راست (حرکت مورب ممنوع است)
 - نقطه شروع: $(0,0)$
 - نقطه پایان: $(4,4)$
- 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:

- شروع از نقطه اولیه $(0,0)$ و افزودن به صف
- بررسی همسایه‌های چهارگانه سلول جاری
- افزودن سلول‌های معتبر به صف
- تکرار فرآیند تا رسیدن به نقطه هدف
- 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:

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

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. این الگوریتم یک روش بهینه‌سازی تکاملی است که از اصول انتخاب طبیعی در نظریه داروین الهام گرفته شده است. هدف این الگوریتم یافتن بهترین راه‌حل برای یک مسئله است با شبیه‌سازی فرآیند تکامل در طبیعت که مراحل آن به صورت زیر می‌باشد:

۱. ایجاد جمعیت اولیه : تولید مجموعه‌ای از افراد تصادفی که راه‌حل‌های ممکن هستند. تابع تناسب
۲. محاسبه میزان تناسب (Fitness Evaluation): هر فرد بر اساس میزان مطلوبیت خود ارزیابی می‌شود
۳. انتخاب والدین (Selection): افراد برتر برای تولید نسل جدید انتخاب می‌شوند.
۴. ترکیب ژنتیکی (Crossover): دو والد ترکیب شده و فرزندان جدید ایجاد می‌شوند.
۵. جهش (Mutation): تغییر تصادفی در برخی ژن‌ها برای حفظ تنوع جمعیت.
۶. تکرار فرآیند: مراحل بالا تکرار می‌شوند تا به یک راه‌حل بهینه برسیم.

ساختار کد:

تابع `create_initial_population`: هدف این تابع، ایجاد جمعیت اولیه با لیست‌های باینری (0 و 1) است، به گونه‌ای که تعداد دقیق یک‌ها برابر `n_ones` باشد.

```
def create_initial_population(pop_size, genome_length, n_ones):  
    population = []  
    for _ in range(pop_size):  
        individual = [0] * genome_length
```



```

ones_indices = random.sample(range(genome_length), n_ones)

for idx in ones_indices:
    individual[idx] = 1
population.append(individual)

return population

```

ابتدا لیست‌هایی شامل صفر ساخته می‌شوند. سپس تعداد دقیقی از یک‌ها به‌صورت تصادفی تنظیم می‌شود. هر فرد نمایانگر یک ترکیب متفاوت از موقعیت‌های نصب پنل خورشیدی است.

تابع `calculate_fitness`: هدف این تابع، محاسبه مجموع میزان تابش خورشیدی در موقعیت‌های انتخاب‌شده است. مقدار `Fitness` هر فرد برابر مجموع میزان تابش در موقعیت‌های انتخاب‌شده است. فقط موقعیت‌هایی که مقدارشان 1 باشد در محاسبه لحاظ می‌شوند.

```

def calculate_fitness(individual, exposures):
    return sum(bit * exposures[i] for i, bit in enumerate(individual))

```

تابع `tournament_selection`: هدف این تابع، انتخاب والدین با روش `Tournament` است. ۳ فرد تصادفی انتخاب شده و فردی که بالاترین `Fitness` را دارد برنده می‌شود. این روش باعث رقابت بین راه‌حل‌ها و انتخاب بهترین‌ها می‌شود.

```

def tournament_selection(population, exposures, k=3):
    contenders = random.sample(population, k)
    return max(contenders, key=lambda ind: calculate_fitness(ind, exposures))

```

تابع `crossover`: در این تابع با هدف تولید فرزندان جدید، ابتدا دو نقطه تصادفی انتخاب شده و بخش میانی والدین با یکدیگر تعویض می‌شود. سپس باید تعداد یک‌ها تنظیم شوند تا دقیقاً برابر `n_ones` باشد

```

def crossover(parent1, parent2):
    length = len(parent1)
    child1, child2 = copy.deepcopy(parent1), copy.deepcopy(parent2)
    p1, p2 = sorted(random.sample(range(1, length-1), 2))
    child1[p1:p2], child2[p1:p2] = parent2[p1:p2], parent1[p1:p2]
    repair(child1)
    repair(child2)
    return child1, child2

def repair(child, n_ones=5):
    while sum(child) > n_ones:
        i = random.choice([i for i, b in enumerate(child) if b == 1])
        child[i] = 0

    while sum(child) < n_ones:
        i = random.choice([i for i, b in enumerate(child) if b == 0])
        child[i] = 1

```

تحليل الكوريتيم:

- پیچیدگی زمانی برابر با $O(\text{generations} \times \text{pop_size} \times \text{genome_length})$ است؛ یعنی افزایش جمعیت باعث افزایش زمان پردازش می‌شود.
- پیچیدگی فضایی برابر با $O(\text{pop_size} \times \text{genome_length})$ است؛ یعنی افزایش جمعیت باعث افزایش مصرف حافظه خواهد شد.

بهینه‌سازی:

برای بهبود عملکرد الگوریتم، می‌توان `pop_size` یا `generations` را کاهش داد، یا اینکه `k` را در تابع `tournament_selection` هوشمندانه انتخاب کرد. همچنین، می‌توان از کتابخانه `numpy` برای بهینه‌سازی در پردازش آرایه‌ها بهره برد (آرایه‌های ساخته شده بوسیله این کتابخانه، از لیست‌های معمولی در پردازش داده‌ها سریع‌ترند و مصرف حافظه را کاهش می‌دهند).

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 شیوه‌ای 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

چالش‌های محاسباتی و الگوریتمی:

در فرایند توسعه، با چالش‌های زیر مواجه شدیم:

۱. 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.
۲. بهینه‌سازی مصرف حافظه: نیاز به ذخیره‌سازی تمام تخصیص‌های موقت در حین بازگشت
۳. 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 T_4 فعال باشد، T_1 حداکثر ۳ واحد انرژی می‌تواند مصرف کند) اشاره کرد.

راهکارهای پیشنهادی:

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

محدودیت‌ها:

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

۱. برای $n > 8$ غیر عملی می‌شود.
۲. پیچیدگی زمانی بسیار بالا $(O(n^2))$
۳. پیچیدگی فضایی نسبتاً مناسب $(O(n))$

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:

```
import random  
import copy
```

```
def create_initial_population(pop_size, genome_length, n_ones):  
    population = []  
    for _ in range(pop_size):  
        individual = [0] * genome_length  
        ones_indices = random.sample(range(genome_length), n_ones)  
        for idx in ones_indices:  
            individual[idx] = 1  
        population.append(individual)  
    return population
```

```
def calculate_fitness(individual, exposures):  
    return sum(bit * exposures[i] for i, bit in enumerate(individual))
```

```
def tournament_selection(population, exposures, k=3):  
    contenders = random.sample(population, k)  
    return max(contenders, key=lambda ind: calculate_fitness(ind, exposures))
```

```
def repair(child, n_ones=5):
```

```

while sum(child) > n_ones:
    i = random.choice([i for i, b in enumerate(child) if b == 1])
    child[i] = 0

while sum(child) < n_ones:
    i = random.choice([i for i, b in enumerate(child) if b == 0])
    child[i] = 1

def crossover(parent1, parent2):
    length = len(parent1)
    child1, child2 = copy.deepcopy(parent1), copy.deepcopy(parent2)
    p1, p2 = sorted(random.sample(range(1, length-1), 2))
    child1[p1:p2], child2[p1:p2] = parent2[p1:p2], parent1[p1:p2]
    repair(child1)
    repair(child2)
    return child1, child2

def mutate(individual, mutation_rate=0.1):
    if random.random() < mutation_rate:
        ones = [i for i, b in enumerate(individual) if b == 1]
        zeros = [i for i, b in enumerate(individual) if b == 0]
        if ones and zeros:
            i1 = random.choice(ones)
            i0 = random.choice(zeros)
            individual[i1], individual[i0] = 0, 1

    return individual

```

```

def genetic_algorithm(exposures, pop_size=10, generations=20):
    population = create_initial_population(pop_size, len(exposures), 5)
    for gen in range(1, generations+1):
        new_pop = []
        elite = max(population, key=lambda ind: calculate_fitness(ind, exposures))
        new_pop.append(elite)
        while len(new_pop) < pop_size:
            p1 = tournament_selection(population, exposures)
            p2 = tournament_selection(population, exposures)
            while p2 == p1:
                p2 = tournament_selection(population, exposures)

            c1, c2 = crossover(p1, p2)
            new_pop.append(mutate(c1))
            if len(new_pop) < pop_size:
                new_pop.append(mutate(c2))

        population = new_pop

    best = max(population, key=lambda ind: calculate_fitness(ind, exposures))
    return best, calculate_fitness(best, exposures)

def main():
    print("=== AstroBot Solar Panel Placement System ===")
    exposures = [8.5, 2.3, 9.1, 4.2, 7.8, 3.1, 6.7, 9.8, 1.5, 5.4]
    print("Exposure list:", exposures)
    best_sol, best_fit = genetic_algorithm(exposures)
    print("\n--- Best Solution ---")

```

```
print("Binary genome:", best_sol)
print(f"Total exposure: {best_fit:.2f}")
```

```
if __name__ == "__main__":
    main()
```

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()

```


<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://www.geeksforgeeks.org/genetic-algorithms/>

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

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

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

[ChatGPT](#)

[Grok](#)