ĐẠI HỌC QUỐC GIA TP.HCM
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

Khoa Khoa học máy tính

Môn học Phân Tích Và Thiết Kế Thuật Toán

Bài tập Backtracking, Bruteforce

Cao Lê Công Thành
MSSV: 23521437
Đặng Quang Vinh
MSSV: 23521786

Mục lục

1 Bài tập 1

1.1 Nguyên lý cơ bản của thuật toán quay lui

Backtracking hoạt động dựa trên nguyên tắc **"thử nghiệm và lùi lại"** (trial and error):

Thử nghiệm các khả năng

- Bắt đầu với một trạng thái hoặc lựa chọn ban đầu.
- Dựa vào từng lựa chọn, tiến hành thử nghiệm từng nhánh của cây tìm kiếm.

Kiểm tra điều kiện

- Nếu một nhánh tìm kiếm không thể dẫn đến lời giải hợp lệ hoặc không tối ưu, thuật toán sẽ lùi lại về trạng thái trước đó.

Lùi lại (Backtrack)

- Sau khi hoàn thành việc thử nghiệm một nhánh, thuật toán lùi lại và thử nhánh khác.

Tiếp tục tìm kiếm

- Quay lại các lựa chọn và lặp lại quá trình thử nghiệm và lùi lại cho đến khi tất cả các khả năng được khám phá hoặc tìm được lời giải.

1.2 Tại sao Backtracking thường được sử dụng để giải bài toán tổ hợp?

Backtracking thường được sử dụng để giải các bài toán tổ hợp vì các bài toán tổ hợp có tính chất có nhiều nhánh lựa chọn và ràng buộc.

Khám phá tất cả các khả năng

Backtracking cho phép thử tất cả các lựa chọn có thể, đảm bảo rằng mọi khả năng đều được khám phá.

Đảm bảo ràng buộc

Trong các bài toán tổ hợp, có thể có nhiều ràng buộc (ví dụ: không thể gán cùng lúc hai việc nếu vi phạm thời gian). Backtracking sẽ dừng ở các nhánh không hợp lệ và lùi lại.

Giảm độ phức tạp với lộ trình lùi lại thông minh

Thay vì thử tất cả các tổ hợp hoàn toàn từ đầu, Backtracking chỉ thử nghiệm từng nhánh của bài toán và quay lui khi cần, làm giảm đáng kể không gian tìm kiếm.

Dễ dàng áp dụng với bài toán dạng cây tìm kiếm

Bài toán tổ hợp thường có thể biểu diễn dưới dạng cây tìm kiếm, và Backtracking rất phù hợp cho việc duyệt qua các nhánh này.

1.3 So sánh điểm khác biệt chính giữa thuật toán nhánh cận (Branch and Bound) và quay lui (Backtracking) khi tìm kiếm lời giải tối ưu

Khi tìm kiếm lời giải tối ưu, cả **Branch and Bound** và **Backtracking** đều là các kỹ thuật tìm kiếm trong không gian trạng thái. Tuy nhiên, chúng có một số điểm khác biệt quan trọng như sau:

Nguyên lý hoạt động

- **Backtracking**: Hoạt động dựa trên nguyên tắc **"thử nghiệm và lùi lại"** (trial and error). Backtracking thử nghiệm tất cả các khả năng trong không gian tìm kiếm và lùi lại khi phát hiện nhánh không hợp lệ hoặc không tối ưu.
- **Branch and Bound**: Sử dụng thông tin giới hạn (bounds) để loại bỏ các nhánh không thể đạt được lời giải tối ưu từ đầu, thay vì thử nghiệm tất cả các nhánh như trong Backtracking.

Kỹ thuật tối ưu hóa

- **Backtracking**: Không áp dụng kỹ thuật cận (bounding). Backtracking thử nghiệm tất cả các khả năng và lùi lại khi phát hiện điều kiện không phù hợp.
- **Branch and Bound**: Áp dụng kỹ thuật giới hạn (bounding) để giảm không gian tìm kiếm. Nghĩa là nếu tổng chi phí hoặc một chỉ tiêu tối ưu vượt quá giới hạn hiện tại, thuật toán sẽ không tiếp tục tìm kiếm trong nhánh đó.

Mục đích và hiệu quả

- **Backtracking**: Thường được sử dụng để giải bài toán tổ hợp, tìm kiếm các lời giải khả thi hoặc khám phá không gian tìm kiếm đầy đủ. Không nhất thiết tìm kiếm lời giải tối ưu.
- **Branch and Bound**: Dùng để tìm kiếm lời giải tối ưu trong không gian trạng thái, tối ưu hóa thông qua việc loại bỏ các nhánh không tiềm năng dựa vào thông tin giới hạn.

Ứng dụng

- **Backtracking**: Thường được áp dụng trong các bài toán tổ hợp, như bài toán 8 Queen, hoán vị, và các bài toán với ràng buộc thử nghiệm trực tiếp.
- **Branch and Bound**: Thường được áp dụng trong các bài toán tối ưu hóa, như bài toán TSP (Traveling Salesman Problem), bài toán lịch trình tối ưu, hoặc bài toán tìm kiếm giải pháp tối ưu với chi phí tối thiểu.

1.4 Trình bày ưu điểm và nhược điểm của phương pháp Brute Force. Tại sao nó thường được xem là phương pháp kém hiệu quả trong các bài toán lớn

Ưu điểm của phương pháp Brute Force

Phương pháp **Brute Force** (tìm kiếm thử tất cả các khả năng) có một số ưu điểm như sau:

- **Đơn giản và dễ hiểu**: Brute Force có cách tiếp cận đơn giản, chỉ cần thử nghiệm tất cả các khả năng hoặc tổ hợp có thể xảy ra và chọn kết quả hợp lệ nhất.
- **Không phụ thuộc vào thông tin đặc biệt**: Không cần hiểu sâu về cấu trúc bài toán hoặc thông tin ràng buộc cụ thể. Điều này giúp Brute Force dễ dàng áp dụng cho hầu hết các bài toán.
- **Đảm bảo tính chính xác**: Vì thử tất cả các khả năng, Brute Force đảm bảo không bỏ qua bất kỳ khả năng hợp lệ nào. Do đó, kết quả cuối cùng sẽ chính xác nếu bài toán đủ nhỏ.

Nhược điểm của phương pháp Brute Force

Dù có một số ưu điểm, Brute Force cũng có các nhược điểm như sau:

- **Thời gian tính toán lớn**: Brute Force cần duyệt qua tất cả các khả năng trong không gian tìm kiếm, dẫn đến thời gian tính toán rất lớn nếu bài toán có không gian tìm kiếm lớn.
- **Không gian bộ nhớ lớn**: Với bài toán có nhiều khả năng hoặc tổ hợp, Brute Force có thể tiêu tốn nhiều bộ nhớ và làm giảm hiệu quả tính toán.
- **Kém hiệu quả với bài toán lớn**: Do phải thử nghiệm tất cả các khả năng, Brute Force nhanh chóng trở nên không khả thi với bài toán có số lượng khả năng lớn hoặc không gian tìm kiếm lớn.

Tại sao Brute Force thường được xem là phương pháp kém hiệu quả trong các bài toán lớn?

Brute Force thường được xem là phương pháp kém hiệu quả trong các bài toán lớn vì các lý do sau:

- **Thời gian tính toán tăng theo cấp số nhân**: Với mỗi bài toán lớn, số khả năng hoặc tổ hợp cần thử nghiệm có thể tăng rất nhanh, thường theo dạng lũy thừa hoặc cấp số nhân. Điều này dẫn đến thời gian tính toán rất lớn và không thể hoàn thành trong thời gian hợp lý.
- **Không thể mở rộng với kích thước bài toán lớn**: Vì không có cơ chế tối ưu hóa hay cận thông minh, Brute Force không thể xử lý bài toán lớn hoặc các bài toán có giới hạn tài nguyên tính toán.
- **Phụ thuộc hoàn toàn vào thử nghiệm brute force**: Không tối ưu hóa qua thông tin bài toán hoặc chiến lược thông minh khác như cận (bounding) hay heuristics.

Kết luận

Mặc dù Brute Force đảm bảo tính chính xác và dễ hiểu, nhưng với bài toán lớn, phương pháp này trở nên không khả thi do thời gian và bộ nhớ tiêu tốn quá nhiều. Vì vậy, các thuật toán tối ưu hóa hoặc các kỹ thuật thông minh như Backtracking, Branch and Bound thường được ưu tiên trong thực tế.

2 Bài tập 2

2.1 Mô tả bài toán

Trò chơi 24 yêu cầu sử dụng bốn giá trị từ các thẻ bài, áp dụng các phép toán số học để tìm giá trị lớn nhất không vượt quá 24. Đặc điểm của bài toán là việc duyệt qua tất cả các tổ hợp giá trị, phép toán và thứ tự tính toán, cùng với kiểm tra điều kiện hợp lệ (kết quả phép chia phải là số nguyên, không ghép số).

2.2 Yêu cầu và ràng buộc

- **Phép toán hợp lệ**: Cộng $(+)$, Trừ $(-)$, Nhân $(\times)$, Chia $(/)$.
- **Sử dụng đủ 4 thẻ bài**: Biểu thức phải bao gồm tất cả giá trị của bốn thẻ bài.
- **Kết quả phép toán trung gian**: Kết quả phép chia phải là số nguyên.
- **Số lượng bộ bài**: Tối đa $N = 5$ bộ bài, mỗi bộ bài gồm 4 giá trị $1 \leq C \leq 13$.

2.3 Phương pháp và thuật toán được sử dụng

2.3.1 Phương pháp tổng quát

Sử dụng phương pháp **tìm kiếm tổ hợp (brute-force exhaustive search)**, đảm bảo kiểm tra tất cả các khả năng sắp xếp giá trị thẻ bài, kết hợp phép toán và cách đặt ngoặc.

2.3.2 Các bước cụ thể

1. **Sinh tất cả các hoán vị của các thẻ bài**: Dùng thuật toán hoán vị (permutation) để tạo ra tất cả $4! = 24$ cách sắp xếp thẻ bài.
2. **Sinh tất cả tổ hợp phép toán**: Với 4 giá trị, có 3 phép toán cần sử dụng. Dùng thuật toán sinh tổ hợp lặp (cartesian product) để tạo ra $4^3 = 64$ tổ hợp phép toán (cộng, trừ, nhân, chia).
3. **Thử các cách đặt dấu ngoặc**: Sử dụng các biểu thức có ngoặc để đảm bảo kiểm tra đúng thứ tự thực hiện phép toán. Có 5 cách đặt dấu ngoặc chính cho 4 giá trị, ví dụ:
   - $((a \circ b) \circ c) \circ d$,
   - $(a \circ (b \circ c)) \circ d$, ...
4. **Tính giá trị biểu thức và kiểm tra điều kiện hợp lệ**:

- Đảm bảo phép chia không tạo ra số thập phân (kể cả giá trị trung gian).
- Kiểm tra xem kết quả có nhỏ hơn hoặc bằng 24 không.
- Lưu lại kết quả lớn nhất hợp lệ.

2.3.3 Thuật toán cụ thể

```
from itertools import permutations, product

def evaluate_expression(a, b, op):
    """Thuc hien phep toan a op b va kiem tra hop le"""
    if op == '+':
        return a + b
    elif op == '-':
        return a - b
    elif op == '*':
        return a * b
    elif op == '/' and b != 0 and a % b == 0:
        return a // b
    return None  # Phep toan khong hop le

def calculate(cards):
    best_result = float('-inf')  # Gia tri lon nhat tim duoc, ban dau la am vo cung

    # Sinh tat ca cac hoan vi cua cac the bai
    for perm in permutations(cards):
        # Sinh tat ca to hop phep toan
        for ops in product(['+', '-', '*', '/'], repeat=3):
            # Thu tat ca cach dat dau ngoac
            expressions = [
                lambda: evaluate_expression(
                    evaluate_expression(
                        evaluate_expression(perm[0], perm[1], ops[0]),
                        perm[2],
                        ops[1],
                    ),
                    perm[3],
                    ops[2],
                ),
                lambda: evaluate_expression(
                    evaluate_expression(perm[0],
                        evaluate_expression(perm[1], perm[2], ops[1]), ops[0]),
                    perm[3],
                    ops[2],
                ),
                lambda: evaluate_expression(
                    perm[0],
```

```
                        evaluate_expression(evaluate_expression(perm[1],
                            perm[2], ops[1]), perm[3], ops[2]),
                        ops[0],
                    ),
                    lambda: evaluate_expression(
                        perm[0],
                        evaluate_expression(perm[1],
                            evaluate_expression(perm[2], perm[3],
                            ops[2]), ops[1]),
                        ops[0],
                    ),
                    lambda: evaluate_expression(
                        evaluate_expression(perm[0], perm[1],
                            ops[0]),
                        evaluate_expression(perm[2], perm[3],
                            ops[2]),
                        ops[1],
                    ),
                ]

                # Duyet qua tung cach bieu dien va cap nhat ket qua
                    lon nhat
                for expr in expressions:
                    try:
                        result = expr()
                        if result is not None and result <= 24:
                            best_result = max(best_result, result)
                    except:
                        pass  # Bo qua cac phep toan khong hop le

    return best_result if best_result != float('-inf') else None

def main():
    # Nhap so luong bo bai
    N = int(input("Nhap so luong bo bai: "))
    results = []

    for _ in range(N):
        # Nhap 4 gia tri the bai
        cards = [int(input()) for _ in range(4)]
        # Tinh gia tri lon nhat khong vuot qua 24
        result = calculate(cards)
        results.append(result)

    # Xuat ket qua
    for res in results:
        print(res)

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

2.4 Ưu điểm và nhược điểm của thuật toán

2.4.1 Ưu điểm

- Đảm bảo duyệt qua toàn bộ không gian tìm kiếm, từ đó tìm ra kết quả tối ưu nhất.
- Dễ triển khai và có thể áp dụng trực tiếp cho bài toán tổ hợp nhỏ (với số lượng thẻ $n = 4$).

2.4.2 Nhược điểm

- **Tính toán dư thừa**: Duyệt toàn bộ tổ hợp dẫn đến chi phí tính toán cao.
- **Độ phức tạp lớn**: Độ phức tạp $O(4! \times 4^3 \times 5)$, tuy nhỏ với $n = 4$, nhưng có thể mở rộng chậm với bài toán lớn hơn.
- **Không tối ưu hóa**: Không tận dụng các quy tắc rút gọn hoặc loại bỏ tổ hợp vô nghĩa từ sớm.

2.5 Kết luận

Phương pháp **tìm kiếm tổ hợp (brute-force exhaustive search)** đảm bảo giải quyết bài toán đúng yêu cầu, nhưng sẽ cần cải tiến thuật toán nếu mở rộng bài toán với không gian tìm kiếm lớn hơn (như tăng số lượng thẻ hoặc phép toán).