Knowledge Representation and Learning UE Assignment 1

Giovanni Filomeno (K12315325), Manu Gupta (K11936702)

April 21, 2024

1 Exercise 1

1.1 Problem Description

Given a list of numbers $L = [n_1, n_2, \ldots, n_k]$ and a target number N, the task is to develop a Prolog predicate named `solve(L, N)`, which identifies a sequence of arithmetic operations (addition, subtraction, multiplication, division) to apply between each pair of adjacent numbers in the list. The objective is for the final evaluated expression to equal N. The sequence of operations should be displayed to show the execution order and the transformations of the list at each step.

1.2 Solution Approach

The solution involves recursively exploring different combinations of arithmetic operations between the consecutive numbers in the list. The approach ensures that each combination is tested for its ability to transform the list into the target number N.

1.2.1 Methodology

1. We defined arithmetic operations using the predicate `apply_op` which applies an operation between two numbers and produces a result.
2. We developed a recursive helper predicate `solve_helper` that attempts to apply every possible operation between the first two numbers in the list, then recursively processes the result with the rest of the list.
3. The code accumulates the sequence of operations and their positions as the recursion unfolds, and backtrack through different possible operations if the current path does not lead to the target N.
4. Once the base case is reached (when the list is reduced to the single element N), the accumulated sequence of operations is confirmed as a solution.

1.3 Implementation

The program is implemented in Prolog as follows:

```
% Operator definitions with evaluation
apply_op(add, X, Y, Z) :- Z is X + Y.
apply_op(sub, X, Y, Z) :- Z is X - Y.
apply_op(mul, X, Y, Z) :- Z is X * Y.
apply_op(div, X, Y, Z) :- Y \= 0, Z is X / Y.

% solve/3 - main predicate
solve(L, N, Ops) :-
    solve_helper(L, N, [], Ops).

% solve_helper/4 - helper predicate
solve_helper([N], N, Ops, Ops).
solve_helper([X, Y | Rest], N, AccOps, Ops) :-
    apply_op(Op, X, Y, Z),
    append(AccOps, [[0, Op]], NewAccOps),
    solve_helper([Z | Rest], N, NewAccOps, Ops).
```

1.4 Output of the Code

The execution of the predicate `solve` with the list `[8,2,3,6,2]` targeting the number 27 yields the following output:

```
solve([8,2,3,6,2], 27).
Output:
[[0,sub],[0,add],[0,mul],[0,div]]
true.
```

The solution corresponds to the arithmetic expression $((8 - 2) + 3) \times 6/2$, which simplifies to 27. This demonstrates how the program systematically applies the operators subtraction, addition, multiplication, and division in sequence to achieve the desired result.

2 Exercise 2

2.1 Problem Description

The task involves writing two Prolog predicates: `compress` and `decompress`. The first predicate compresses a list of numbers where segments of identical numbers are replaced with a list containing the number and its frequency if the segment's length is greater than two. Otherwise, the numbers are left unchanged. The second predicate decompresses the list back to its original form.

2.2 Solution Approach

Compression

The compression algorithm iterates over the list, counting occurrences of consecutive identical numbers and encoding them as specified:

- If the segment length is greater than 2, it is encoded as `[number, length]`.
- Segments of length 2 are kept as two consecutive numbers.
- Single numbers are left unchanged.

Decompression

The decompression process reverses the encoding by expanding each encoded segment back to its original sequence of numbers based on the stored number and count.

2.3 Implementation

Prolog Code for Compression and Decompression

```
% Compression helper predicates
compress(List, CompressedList) :-
    compress_helper(List, CompressedList, 1).

compress_helper([], [], _).
compress_helper([X], [[X, Count]], Count) :- Count > 2.
compress_helper([X], [X, X], 2).
compress_helper([X], [X], 1).
compress_helper([X, Y | Rest], Compressed, Count) :-
    X \= Y,
    (   Count > 2
    ->  Compressed = [[X, Count] | RestCompressed]
    ;   Count = 2
    ->  Compressed = [X, X | RestCompressed]
    ;   Compressed = [X | RestCompressed]),
    compress_helper([Y | Rest], RestCompressed, 1).
compress_helper([X, X | Rest], Compressed, Count) :-
    NewCount is Count + 1,
    compress_helper([X | Rest], Compressed, NewCount).

% Decompression helper predicates
decompress(CompressedList, List) :-
    decompress_helper(CompressedList, List).

decompress_helper([], []).
```

```
decompress_helper([[N, C] | T], List) :-
    C > 2,
    length(Full, C),
    maplist(=(N), Full),
    decompress_helper(T, Rest),
    append(Full, Rest, List).
decompress_helper([X | T], [X | List]) :-
    decompress_helper(T, List).
```

2.4 Output of the Code

The Prolog predicates `compress` and `decompress` demonstrate the list manipulation capabilities of the program. Given a list with contiguous sequences of integers, `compress` produces a compressed list where sequences longer than two elements are represented by their start and end values, while shorter sequences are left unchanged. The `decompress` function then restores the original list from this compressed format.

The commands and their outputs are shown below:

```
compress([2,2,2,3,3,3,4,4,5,5,5,5,6,6,6,6,6], CompressedList).
decompress([[2, 3], [3, 3], 4, 4, [5, 4], [6, 5]], L).
Output:
Compressed:
[[2,3],[3,3],4,4,[5,4],[6,5]]
Decompressed:
[2,2,2,3,3,3,4,4,5,5,5,5,6,6,6,6,6]
true.
```

The compressed output therefore encodes longer sequences as ranges, reducing the list's complexity, while the decompressed output accurately shows the original list, confirming the correctness of both the `compress` and `decompress` predicates.

3 Exercise 3

3.1 Problem Description

The exercise involves writing two Prolog predicates: `compress` and `decompress`. The first predicate compresses a list of integers, where segments of continuous integers are encoded as ranges if the segment length is greater than two. Otherwise, the segment is kept unchanged. The second predicate decompresses the list back to its original form.

3.2 Solution Approach

3.2.1 Compression

The compression algorithm iterates over the list, counting occurrences of consecutive continuous numbers and encoding them as specified:

- If the segment length is greater than 2, it is encoded as `[start, end]` where 'end' is inclusive.
- Segments of length 2 are kept as two consecutive numbers.
- Single numbers are left unchanged.

3.2.2 Decompression

The decompression process reverses the encoding by expanding each encoded segment back to its original sequence of numbers.

3.3 Implementation

3.4 Prolog Code for Compression and Decompression

```
% Compression helper predicates
compress_helper([], [], _Start, _End, _LastAdded).
compress_helper([H|T], Compressed, Start, End, LastAdded) :-
    Next is End + 1,
    (   H == Next
    ->  compress_helper(T, Compressed, Start, H, LastAdded)
    ;   Length is End - Start + 1,
        (   Length > 2
        ->  NewSegment = [[Start, End]]
        ;   Length == 2
        ->  NewSegment = [Start, End]
        ;   NewSegment = [Start]
        ),
        compress_helper(T, Rest, H, H, H),
        append(NewSegment, Rest, Compressed)
    ).

compress([H|T], Compressed) :-
    compress_helper(T, Compressed, H, H, H).
compress([], []).

% Decompression helper predicates
expand_range(Start, End, List) :-
    findall(X, between(Start, End, X), List).
```

```
decompress_helper([], []).
decompress_helper([H|T], List) :-
    (   is_list(H)
    ->  H = [Start, End],
        expand_range(Start, End, Expanded),
        decompress_helper(T, Rest),
        append(Expanded, Rest, List)
    ;   decompress_helper(T, Rest),
        List = [H|Rest]
    ).

decompress(Compressed, List) :-
    decompress_helper(Compressed, List).
```

3.5 Output of the Code

The Prolog predicates `compress` and `decompress` handle the transformation of lists of continuous integers into a compressed format and back to the original list format. The `compress` function encodes segments of continuous integers longer than two as a range from start to end, and the `decompress` function expands these ranges back into their original continuous sequences.

The execution of these functions and their outputs are presented below:

```
compress([1,2,4,5,6,7,8,10,15,16,17,20,21,22,23,24,25], CompressedList).
decompress([1, 2, [4, 8], 10, [15, 17], [20, 25]], L).
Output:
Testing compression:
[1,2,[4,8],10,[15,17],[20,25]]
Testing decompression:
[1,2,4,5,6,7,8,10,15,16,17,20,21,22,23,24,25]
true.
```

The compressed output efficiently reduces the representation of longer continuous sequences using ranges, thus simplifying the data without loss of information. The decompressed output accurately reconstructs the original sequence of integers, verifying the effectiveness of the compression and decompression logic implemented in Prolog.

4 Conclusion

By exploring Prolog with these 3 exercises, we learned about its capabilities for logical reasoning, pattern matching, and effective recursion. These exercises demonstrated Prolog's strength in handling complex list manipulations and data transformations. Through recursive problem solving and the strategic use of Prolog's logical programming features, we gained a nuanced understanding of constructing efficient algorithms.