Islamic University of Technology

Artificial Intelligence Lab

CSE 4712

Lab 5

Author:
Ishmam Tashdeed
CSE, IUT

Contents

1 Tasks

1.1 Task 1

Write a predicate `shortest_path(Start, End, Path, Cost, MaxEdge)` that avoids cycles and finds the shortest path from `Start` to `End`. The predicate should return `Path` as a list of nodes, `Cost` as the total sum of edge weights along the path, and `MaxEdge` as the largest weight among the edges in the path. If multiple paths have the same minimum total cost, the predicate should select the path whose maximum edge weight is smallest. Represent the graph using facts of the form `edge(Node1, Node2, Weight)`.

The edges and their cost can be given as follows:

```
edge(a, b, 3).
edge(a, c, 5).
edge(b, c, 2).
edge(b, d, 4).
edge(c, d, 1).
```

Sample Input:

```
shortest_path(a, d, Path, Cost, MaxEdge).
```

Sample Output:

```
Cost = 6,
MaxEdge = 3,
Path = [a, b, c, d].
```

Here, both paths [a,c,d] and [a,b,c,d] have a cost of 6, but the maximum edge cost along the former path is 5, whereas it is 3 for the latter. Hence, the latter path is chosen.

1.2 Task 2

Write a predicate `shortest_path_food(Start, End, Path, Cost)` that finds the shortest path from `Start` to `End` in a weighted directed graph, while ensuring that each node along the path has a unique type of food. The predicate should return `Path` as a list of nodes and `Cost` as the total sum of edge weights along the path. If it is not possible to reach the destination while satisfying the food uniqueness constraint, it should return `False`. Represent the graph using facts of the form `edge(Node1, Node2, Weight)` and assign food to each node using facts of the form `food(Node, Food)`.

The edge cost and food associated with each destination can be given as follows:

```
edge(a, b, 3).
edge(a, c, 5).
edge(b, c, 2).
edge(b, d, 4).
edge(c, d, 1).

food(a, sushi).
food(b, pizza).
food(c, sushi).
food(d, burger).
```

Sample Input:

```
shortest_path_food(a, d, Path, Cost).
```

Sample Output:

```
Cost = 7,
Path = [a, b, d].
```

Here, the path [a,b,d] is chosen because the nodes along this path serve different foods: `a` has sushi, `b` has pizza, and `d` has burger. Other paths such as [a,c,d] or [a,b,c,d] are not valid: for [a,c,d], both `a` and `c` have sushi, and for [a,b,c,d], `a` and `c` again both have sushi. Even though the path [a,c,d] has a lower total cost of 6, it is not chosen because it violates the uniqueness of food constraint.

1.3 Task 3

Write a predicate `change(Amount, Coins, NumCoins)` that finds a valid sequence of coin values to give change for a given `Amount` using the given coin values. The predicate should return `Coins` as a list of coin values used in a valid order and `NumCoins` as the total number of coins in the sequence.

The solution must satisfy the following constraints:

- The sum of the coin values equals the required `Amount`.
- No two consecutive coins can have the same value.
- If multiple valid sequences exist, the predicate should return the sequence with the maximum number of coins.

The given coin values can be as follows:

```
coins([1,5,10,50]).
```

Sample Input:

```
change(18, Coins, NumCoins).
```

Sample Output:

```
Coins = [1, 5, 1, 5, 1, 5],
NumCoins = 6.
```

Here, another valid sequence for 18 is `[1,5,1,10,1]` with only 5 coins. Since the predicate returns the sequence with the maximum number of coins, the first sequence `[1,5,1,5,1,5]` is chosen.

Sample Input:

```
change(8, Coins, NumCoins).
```

Sample Output:

```
false
```

The amount 8 cannot be made without repeating the same coin consecutively. For example, a sequence like `[1,5,1,1]` sums to 8, but the last two coins are both 1, which violates the no consecutive coins rule. Hence, the query returns `false`.

1.4 Task 4

Write a predicate `rank_students(RankedList)` that ranks students based on their marks. Each student has marks in 3 different subjects. The ranking rules are:

- A student with a higher lowest grade ranks above a student with a lower lowest grade.
- If two students have the same lowest grade, the student with higher total marks ranks higher.

The predicate should return `RankedList` as a list of entries `[LowestGrade, TotalMarks, Name]`, sorted from highest to lowest rank. A sample student list and their marks can be given as follows:

```
student(alice, 100, 70, 90).
student(bob, 80, 80, 80).
student(charlie, 100, 50, 100).
student(diana, 70, 70, 70).
```

The following grade mapping is written using a conditional expression without the `cut` operator. You are required to **rewrite it** (simply adding a cut is not sufficient, restructuring is needed) as a set of separate Prolog clauses for each grade (`A`, `B`, `C`, `D`, `F`). Include the `cut (!)` wherever necessary so that once a mark matches a grade, Prolog will not backtrack to other grade clauses.

```
% Grading Logic without Cut
subject_grade(Marks, Grade) :-
    ( Marks >= 90 -> Grade = 'A'
    ; Marks >= 80 -> Grade = 'B'
    ; Marks >= 70 -> Grade = 'C'
    ; Marks >= 60 -> Grade = 'D'
    ; Grade = 'F'
    ).
```

.

Sample Input:

```
rank_students(R).
```

Sample Output:

```
R = [['B', 240, bob],
     ['C', 260, alice],
     ['C', 210, diana],
     ['F', 250, charlie]].
```

Here, the ranking is determined first by the lowest grade of each student ($B > C > F$) and then by total marks if the lowest grades are equal. Charlie has an F in the 2nd subject, so he is ranked last despite high total marks.