Report 5

Sobota Dominik, Mierzwa Michael

December 5th, 2016.

Table of Contents

1. Introduction

Report 5 is comprised of

- Frame representation of expert knowledge
- Compute evidence set for Bayesian networks
- Compute Bayesian network probabilities
- Represent Boolean and Multi-state Bayesian networks

1.1 Task Division

The Task Division is a description of the work divided between the two authors of the report as well as the programming contributions that each of them made.

Michael:

- Frames and Test Cases for report_1.pl
- Report sections 1,2,3,5

Dominik:

- Predicates for report_1.pl
- Report sections 2,3,4

2. Task 2.1

2.1 Frame Implementation

The frames that were implemented as per the specification where each attribute value pair was added as described. The additional predicates added were for type_of and instance_of. All of the class names listed in the class section were given a type_of relation as per spec. E.g. trapezoid is a type_of four_sided_polygon and four_sided_polygon is a type of figure.

Where the value for an attribute was “execute function X” we proceeded to use the name of the function that was to be called and later convert it using the univ operator. These functions are detailed in section 2.1.1.

Classes		
Class Name	**Attributes**	**Value**
figure	type_of	Execute function type_of
four_sided_polygon	boundary_length	Execute function sum_of_side_lengths
	area	Execute function four_sided_area
	side_count	4
trapezoid	NONE	
parallelogram	NONE	
rectangle	NONE	
regular_polygon	boundary_length	Execute function side_length_x_side_count
eq_triangle	area	Execute function eq_triangle_area
	side_count	3
square	area	Execute function square_area
	side_count	4
pentagon	area	Execute function pentagon_area
	side_count	5
Instances		
Instance Name	**Attribute**	**Value**
eq_triangle_1	side_ length	A number
square_1	side_ length	A number
pentagon_1	side_ length	A number
parallelogram_1	side_length	List of 4 lengths
	height	A number
rectangle_1	side_length	List of 4 lengths
	height	A number
trapezoid_1	side_length	List of 4 lengths
	height	A number

Table 1: Attribute-value pairs. Instances are to have the given names

2.1.1 Math Functions

The first function created for the class hierarchy was four_sided_area predicate used by all four_sided_polygons. This function was quite tricky since we had implemented the previous ares functions for each shape, but the specification stated that these four_sided_polygons were to not have there own area functions but instead use four_sided_area

```
four_sided_area([Bot,Left,Top,_], Height, A) :- (Height = Left, A is Bot * Height, !;
                                                 (Bot = Top, A is Bot * Height, !;
                                                 C is ((Top + Bot)/2), A is C * Height, !)).
```

Based off the four given side lengths it determines the area of the shape. The trick is that it uses a different formula depending on the proportions of the side_lengths.

The first case makes sure that the left side is equal to the height, this means that out of the three shape (rectangle, trapezoid, parallelogram) only rectangle can fit that description, meaning area is just base times height.

The second case looks if the bot is equal to the top, this means that it cannot be a trapezoid, leaving only parallelogram which has the same calculation of base times height.

The final case means that the shape must be a trapezoid, where the area is (top + bot)/2 * height.

The next function simply adds the side lengths of a four sided shape and returns the sum.

```
sum_of_side_lengths([A,B,C,D], Sum) :- Sum is A + B + C + D.
```

The next function is side_length_x_side_count takes in the side count and side length and multiplies them to create the result

```
side_length_x_side_count(SideLength, SideCount, Result) :-
                                Result is SideLength * SideCount.
```

2.1.1 Math Functions

The next couple functions are simple math ones that don't need explanations of their own, they simply take in side length and return area for either a equilateral triangle, a square, or a pentagon.

2.2 Helper Predicates

The first predicate is 'parent' which was modified from framesQuery.pl to now use type_of as an attribute instead of kind_of. It functions the same but now works for the current implementation of the hierarchy.

```
parent(Frame, ParentFrame) :-
    (Query =.. [Frame, type_of, ParentFrame]
    ;
    Query =.. [Frame, instance_of, ParentFrame]
        ),
    Query.
```

The second helper method is a specialization of parent called superParent. This predicate takes a frame and an attribute and looks for the first parent to have that attribute and returns false if it cannot find said parent. This method was created for the problems we ran into with the rectangle instance and how it's direct parent did not have the area predicate it needed.

```
superParent(Frame, Attr, SuperFrame) :- parent(Frame, A),
                                        Query =.. [A, Attr, _], (Query, SuperFrame = A;
                                        superParent(A,Attr,SuperFrame)), !.
```

The final helper is simply a sort of weird base case for super parent, it is a definition of type_of:

```
    type_of(type_of,_) :- false.
```

This rule simply makes it so that when superParent reaches figure with a type_of type_of and it tries to call the type_of function, it knows that the attribute it is looking for cannot exist.

2.3 side_count

The side_count predicate simply finds the side_count of a given shape and returns it.

```
side_count(Thing, SideCount) :- superParent(Thing, side_count, Parent),
                                Query =.. [Parent, side_count, SideCount], Query,!.
```

First it calls super parent to find where it can find a parent with the side_count attribute. Then using univ it creates a query and calls it to return the side count.

2.4 boundary_length

boundary_length is similar in nature to side_count but differs in the fact that it has an if branch to properly instantiate the boundary length functions of four_sided and regular polygons.

```
boundary_length(Thing, Length) :-
                Side =..[Thing, side_length, SideLength], Side,
                superParent(Thing, boundary_length, Parent),
                Bound =..[Parent,boundary_length,BoundFunc],
                Bound,
                (is_list(SideLength),
                Find =.. [BoundFunc,SideLength,Length], Find
                ;
                side_count(Thing,SideCount),
                Find =.. [BoundFunc,SideLength,SideCount,Length],
                Find ),!.
```

First boundary length searches for the side_length of the given instance, and then tries to find a parent with the boundary_length functor name. Now depending on whether the shape, it either calls regular polygon's boundary_length, or four sided polygons's boundary_length, and because they have different arguments for their predicates, we need to have a branching if to properly deal with both.

The first if branch is for four_sided_polygon's, we know they are four_sided_polygons because their side_length is in a list. We then use Univ to create and then call a query to get the length. Similarly for regular_polygons we create and call a query, but we also need side count as an added argument to find our length.

2.5 area

Area is implemented similarly to boundary_length, the main difference is that instead of having a branching if, it has two versions of the rule, one for regular polygons and one for four sided polygons.

%this rule is for all instances of regular_polygon
```
area(Thing, Area):- Side =.. [Thing, side_length, SideLength], Side
                    , parent(Thing,Parent), Query =.. [Parent, area,
                    AreaFunc], Query,
                    Find =.. [AreaFunc, SideLength, Area], Find, !.
```

%this rule is for all instances of four_sided_polygon
```
area(Thing, Area):-  Side =.. [Thing, side_length, SideLength], Side,
                    High =.. [Thing, height, Height], High,
                     superParent(Thing, area, Parent), Query =.. [Parent,
                    area, AreaFunc], Query,
                    Find =.. [AreaFunc, SideLength, Height,Area], Find, !.
```

.The first rule takes the side length of the instance of a shape and then asks the parent class if there is side_length it can use, it then asks that same parent for the area function and calls it using univ and our newly found side length.

The second rule does the same but this time uses superParent. First the rule gets side_length and height from the instance of the shape, then calls superParent(Thing, area, Parent) to return the first parent with an area function. It then passes over it's values to said function to determine area.

2.6 Test Cases

The test cases simply assert that the area, boundary_length, and side_count predicates work for our predefined shape instances. Six rules to test each predicate, one for each shape.

3. Task 2.2

3.1 Explanation

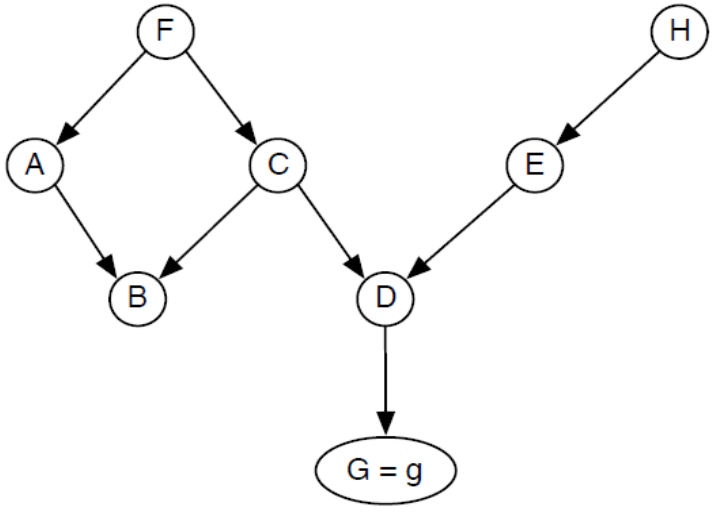

Figure 1: Bayesian network for task 2.2

The evidence sets that d-separate A and H are listed below, and each of the triples along the path from A to H follow three rules to determine if they are blocked

1. If the green node(v) is selected as an evidence node, this path through this triple is then blocked if the v has both tails

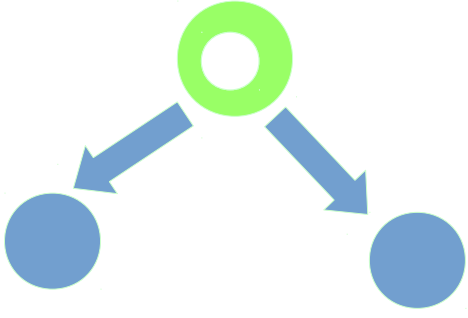

Figure 2: First rule of a blocked path

2 If the green node(v) is selected as an evidence node, this path through this triple is then blocked if the v has both connections has a tail and a tip.

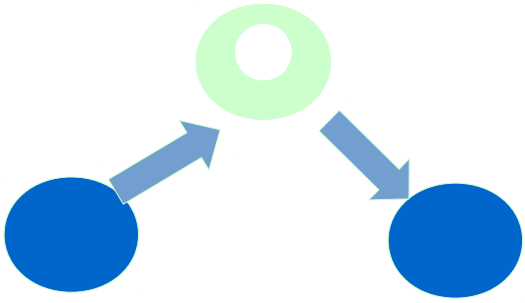

Figure 3: Second rule of a blocked path

3. If the green node(v), or any descendant of v is **not** selected as an evidence node, the path through this triple is blocked if v has both connections as a tip.

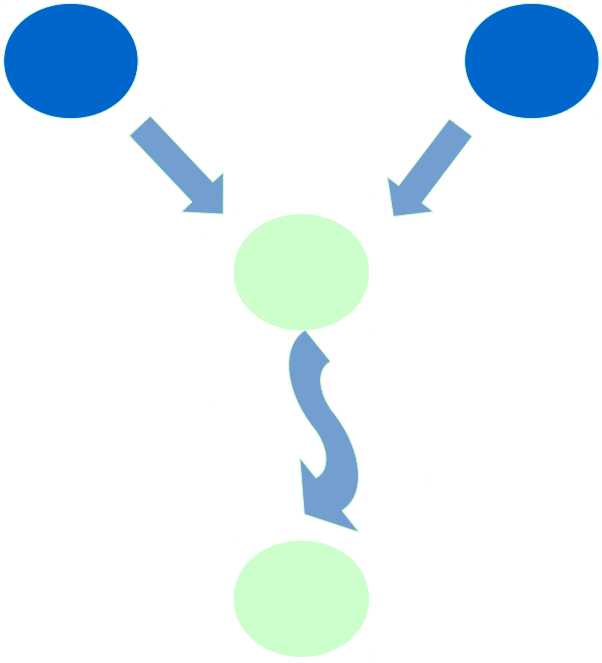

Figure 4: Rule 3 of a blocked path

The evidence sets that d-separate nodes A and H of the Bayesian network described in figure 1 are listed below, with G already being apart of the evidence set. The evidence nodes are placed in such a way that a path from A to H will be blocked by one of these nodes

- {G}
- {G,D}
- {G,E}
- {G,F}
- {G,C}
- {G,E,B}
- {G,C,B}
- {G,F,D}
- {G,C,D}
- {G,E,D}
- {G,F,C}
- {G,F,E}
- {G,C,E}
- {G,F,C,D}
- {G,F,E,D}
- {G,C,E,D}
- {G,B,C,D}
- {G,B,E,D}

4. Task 2.3

4.1 Explanation

Task 2.3 was to create a model representation of the Bayesian network shown below in figure 4 with the algorithm described in f16_4_BayesianNet.pl, The probabilities in table 2 were programmed in report_3.pl

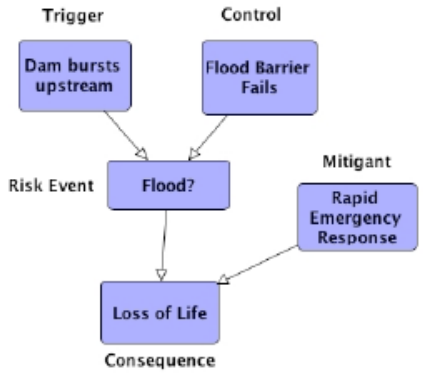

Figure 4: Boolean Bayesian network

Dam Bursts	
No	90%
Yes	10%

Flood Barrier Fails	
No	90%
Yes	10%

Rapid Emergency Response	
No	10%
Yes	90%

Flood?				
Dam Bursts	No		Yes	
Flood Barrier Fails	No	Yes	No	Yes
No	100%	80%	90%	0%
Yes	0%	20%	10%	100%

Loss of Life				
Flood?	No		Yes	
Rapid Emergency Response	No	Yes	No	Yes
No	100%	100%	0%	90%
Yes	0%	0%	100%	10%

Table 2: Bayesian network probabilities

The probabilities we were supposed to calculate and compare were
a) p(flood barrier not failing | flood)
b) p(dam burst | loss of life)

4.2 Comparison of Probabilities

a) p(~flood_barrier_fails| flood)

p(flood_barrier_fails) = 0.1
p(~flood_barrier_fails) = 0.9
p(dam_burst) = 0.1
p(~dam_burst) = 0.9

p(flood) =
  p(flood|dam_burst ^ flood_barrier_fails)*p(dam_burst)*p(flood_barrier_fails)
+ p(flood|dam_burst ^ ~flood_barrier_fails)*p(dam_burst)*p(~flood_barrier_fails)
+ p(flood|~dam_burst ^ flood_barrier_fails)*p(~dam_burst)*p(flood_barrier_fails)
+ p(flood|~dam_burst ^ ~flood_barrier_fails)*p(~dam_burst)*p(~flood_barrier_fails)

p(flood) = 1(0.1)(0.1) + (0.1)(0.1)(0.9) + (0.2)(0.9)(0.1) + 0 = 0.037

p(flood| ~flood_barrier_fails) =
  p(flood|~flood_barrier_fails ^ dam_burst)*p(dam_burst)
+ p(flood|~flood_barrier_fails ^ ~dam_burst)*p(~dam_burst)

p(flood|~flood_barrier_fails) = (0.1)(0.1) + 0 = 0.009

p(~flood_barrier_fails| flood) = p(~flood_barrier_fails) * (p(flood| ~flood_barrier_fails) / p(flood))

p(~flood_barrier_fails| flood) = (0.9) * ((0.001)/(0.037)) = 0.243

The program calculated 0.2342, and this manual calculation gave a value of 0.243, which is pretty much the same result.

b) p(dam_burst| loss_of_life)

p(dam_burst| loss_of_life) = p(dam_burst) * (p(loss_of_life| dam_burst) / p(loss_of_life))

p(dam_burst) = 0.1
p(~dam_burst) = 0.9
p(flood) = 0.037
p(rapid_emergency_response) = 0.9
p(~rapid_emergency_response) = 0.1

p(loss_of_life) =
p(loss_of_life|flood rapid_emergency_response)*p(flood)*p(rapid_emergency_response)
+ p(loss_of_life|flood ^ ~rapid_emergency_response)*p(flood)*p(~rapid_emergency_response)
+ p(~loss_of_life|flood ^ rapid_emergency_response)*p(~flood)*p(rapid_emergency_response)
+ p(~loss_of_life|flood ^ ~rapid_emergency_response)*p(~flood)*p(~rapid_emergency_response)

p(loss_of_life) = 0.1(0.037)(0.9) + (1)(0.037)(0.1) + 0 + 0 = 0.00703

p(loss_of_life|flood) =
p(loss_of_life|flood ^ rapid_emergency_response)*p(rapid_emergency_response)
+ p(loss_of_life| flood ^ ~rapid_emergency_response)*p(~rapid_emergency_response)
= 0.1(0.9) + (1)(0.1) = 0.19

p(flood|dam_burst) =
p(flood|dam_burst ^ flood_barrier_fails)*p(flood_barrier_fails)
+ p(flood|dam_burst ^ ~flood_barrier_fails)*p(~flood_barrier_fails)
= 1(0.1) + 0.1(0.9) = 0.19

p(loss_of_life|~flood) =
p(loss_of_life|~flood ^ rapid_emergency_response)*p(rapid_emergency_response)
+ p(loss_of_life| ~flood ^ ~rapid_emergency_response)*p(~rapid_emergency_response)
= 0 + 0
= 0

p(~flood|dam_burst) =
p(~flood|dam_burst ^ flood_barrier_fails)*p(flood_barrier_fails)
+ p(~flood|dam_burst ^ ~flood_barrier_fails)*p(~flood_barrier_fails)
= 0 + 0
= 0

p(loss_of_life| dam_burst) =
p(loss_of_life|flood)*p(flood|dam_burst)
+ p(loss_of_life|~flood)*p(~flood|~dam_burst)
= 0.19(0.19) + 0
= 0.036

p(dam_burst| loss_of_life) =
p(dam_burst) * (p(loss_of_life| dam_burst) / p(loss_of_life))
= (0.1)((0.0361)/(0.00703))
= 0.513

The answer calulcated by the program is 0.5135135... so compared to our manulaly calcultated 0.513, its pretty much the same result.

5. Task 2.4

5.1 Explanation.

Task 2.4 asked us to program a model representation of the multi-state Bayesian network shown in figure 5 with the probabilities defined in table 3. The program uses the algorithm defined in bayesianNet_multistate.pl to create the representation, which is defined in report_4.pl

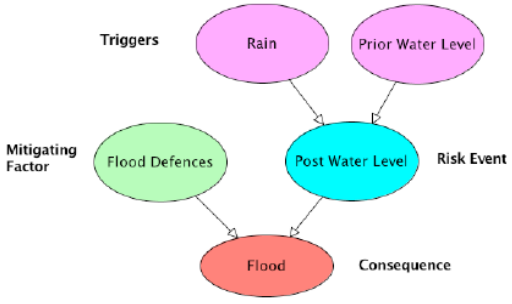

Figure 7: River flooding Bayesian network.

Rain	
None	16.67%
Low	33.33%
High	50%

Prior Water Level	
Low	33.33%
Medium	33.33%
High	33.33%

Flood Defenses	
Poor	16.67%
Good	33.33%
Excellent	50%

Post Water Level									
Prior Water level	Low			Medium			High		
Rain	None	Low	High	None	Low	High	None	Low	High
Low	100%	90%	20%	70%	10%	0%	10%	0%	0%
Medium	0%	10%	70%	30%	80%	20%	70%	10%	0%
High	0%	0%	10%	0%	10%	80%	20%	90%	100%

Flood									
Post Water level	Low			Medium			High		
Flood Defenses	Poor	Good	Excellent	Poor	Good	Excellent	Poor	Good	Excellent
No	85%	95%	98%	60%	80%	92%	35%	65%	85%
Yes	15%	5%	2%	40%	20%	8%	65%	35%	15%

Table 3: Bayesian network probabilities

The probability we were asked to solve is described in the next section p(flood=yes|post_water_level=high)

5.2 Comparison of Probabilities

p(flood=yes|post_water_level=high ^ flood_defenses=poor) = 0.65
p(flood=yes|post_water_level=high ^ flood_defenses=good) = 0.35
p(flood=yes|post_water_level=high ^ flood_defenses=excellent) = 0.15

p(flood_defenses=poor) = 0.1667
p(flood_defenses=good) = 0.3333
p(flood_defenses=excellent) = 0.5

p(flood=yes|post_water_level=high) =
p(flood=yes|post_water_level=high ^ flood_defenses=poor)*p(flood_defenses=poor)
+ p(flood=yes|post_water_level=high ^ flood_defenses=good)*p(flood_defenses=good)
+ p(flood=yes|post_water_level=high ^flood_defenses=excellent)*p(flood_defenses=excellent)
= (0.65)(0.1667) + (0.35)(0.3333) + (0.15)(0.5)
= 0.299

The answer we calculate was 0.299, as compared to 0.300 by the program report_3.pl

6. Task 2.5

6.1 Explanation.

The first difference between, the two state and the multistate bayesian networks is that the two state version has an extra 'prob' predicate. This predicate is used to define the negation of an event, in other words false. A multistate is not composed of true or false, so it has no such rule. Instead, the multistate version uses a helper method hasNode to check if any condition other than X is true. This also means that X is false (given some other state is true) which makes this have the same function as the two state version.

Also note that in multistate, the hasNode predicate is defined to return true if a specific node is in an EventList.

The next difference is in the predecessor predicate. First, in a two state bayesian network, an event is divided into it being true or false, and false being represented with the '~' symbol. This symbol is defined at the top as an operator and is used in the two state accordingly. Multistate however, does not, and this leads them to use a different method for it's states. It store it's events as 'Name=State" pairs. The predecessor predicate has extra rules in the multistate version to interpret these pairs. It uses the univ operator to break up the pairs separated by '=' and uses these values in the exact same way that two state predecessor did.

In conclusion, the only real differences between two state and multistate bayesian networks is that they handle their connective operators differently, and depending on the implementation need a few extra lines to translate from one interpretation back to the original.

End of Report

.