UNIVERSITY OF
CAMBRIDGE
Department of Computer
Science and Technology

Automating representation change across domains for reasoning

Aaron Stockdill
Mateja Jamnik
Daniel Raggi
Peter Cheng
Grecia Garcia Garcia
Holly Sutherland

UNIVERSITY OF
CAMBRIDGE
Department of Computer
Science and Technology

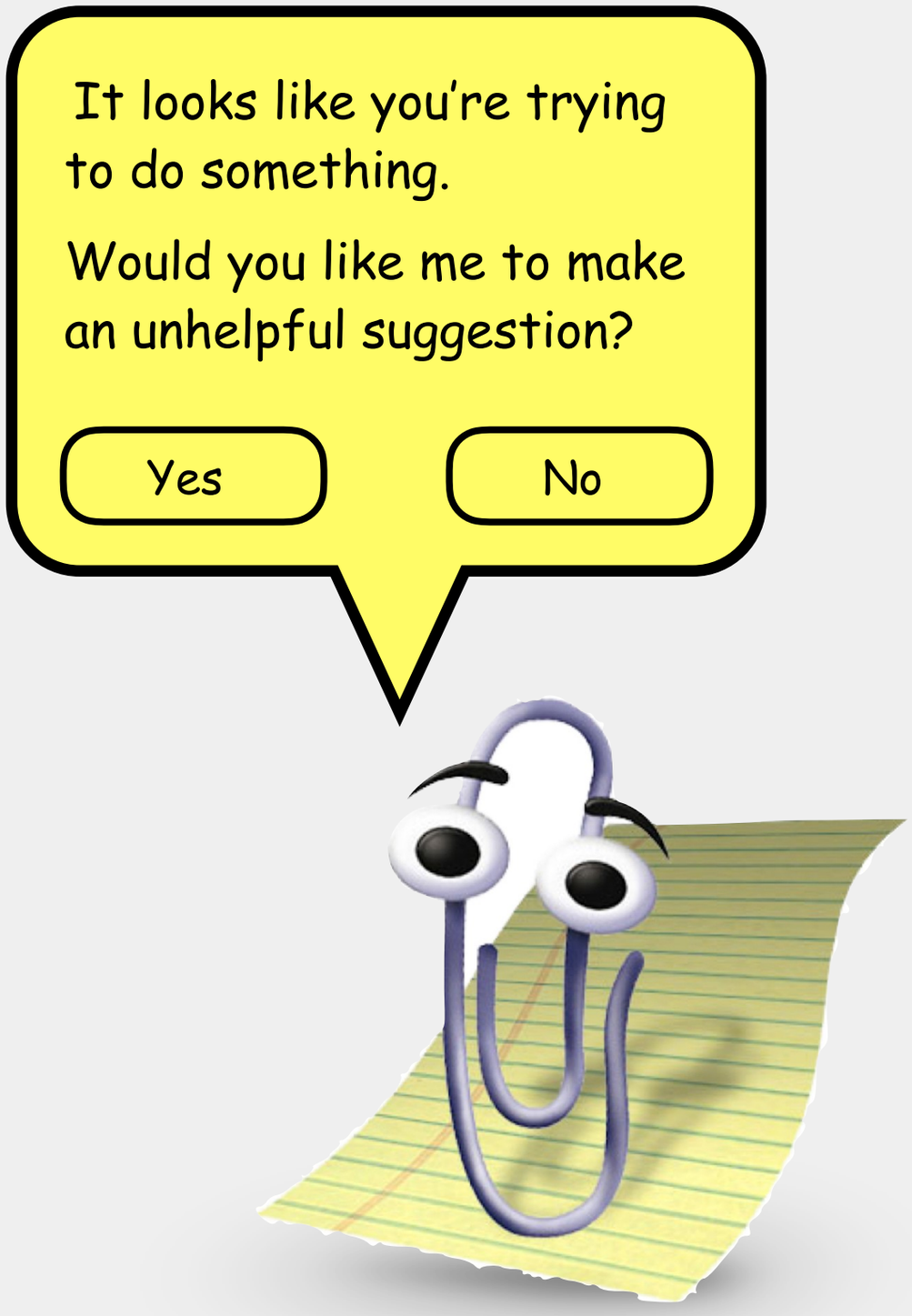
Automating representation change across domains for reasoning

Aaron Stockdill *
Mateja Jamnik
Daniel Raggi
Peter Cheng
Grecia Garcia Garcia
Holly Sutherland

UNIVERSITY OF
CAMBRIDGE
Department of Computer
Science and Technology

Automating representation change across domains for reasoning

Aaron Stockdill
Mateja Jamnik *
Daniel Raggi
Peter Cheng
Grecia Garcia Garcia
Holly Sutherland


It looks like you're trying to do something.
Would you like me to make an unhelpful suggestion?
Yes
No

AutoSave OFF | Financial Sample | Search Sheet

Home Insert Draw Page Layout Formulas Data Review View | Share | Comments

AK663

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Segment	Country	Product	Discount Band	Units Sold	Manufacturin	Sale Price	Gross Sales	Discounts	Sales	COGS	Profit	Date	Month Number	Month Name	Year
2	Government	Canada	Carretera	None	1618.5	$ 3.00	$ 20.00	$ 32,370.00	$ -	$ 32,370.00	$ 16,185.00	$ 16,185.00	01/01/2014	1	January	2014
3	Government	Germany	Carretera	None	1321	$ 3.00	$ 20.00	$ 26,420.00	$ -	$ 26,420.00	$ 13,210.00	$ 13,210.00	01/01/2014	1	January	2014
4	Midmarket	France	Carretera	None	2178	$ 3.00	$ 15.00	$ 32,670.00	$ -	$ 32,670.00	$ 21,780.00	$ 10,890.00	01/06/2014	6	June	2014
5	Midmarket	Germany	Carretera	None	888	$ 3.00	$ 15.00	$ 13,320.00	$ -	$ 13,320.00	$ 8,880.00	$ 4,440.00	01/06/2014	6	June	2014
6	Midmarket	Mexico	Carretera	None	2470	$ 3.00	$ 15.00	$ 37,050.00	$ -	$ 37,050.00	$ 24,700.00	$ 12,350.00	01/06/2014	6	June	2014
7	Government	Germany	Carretera	None	1513	$ 3.00	$ 350.00	$ 529,550.00	$ -	$ 529,550.00	$ 393,380.00	$ 136,170.00	01/12/2014	12	December	2014
8	Midmarket	Germany	Montana	None	921	$ 5.00	$ 15.00	$ 13,815.00	$ -	$ 13,815.00	$ 9,210.00	$ 4,605.00	01/03/2014	3	March	2014
9	Channel Partners	Canada	Montana	None	2518	$ 5.00	$ 12.00	$ 30,216.00	$ -	$ 30,216.00	$ 7,554.00	$ 22,662.00	01/06/2014	6	June	2014
10	Government	France	Montana	None	1899	$ 5.00	$ 20.00	$ 37,980.00	$ -	$ 37,980.00	$ 18,990.00	$ 18,990.00	01/06/2014	6	June	2014
11	Channel Partners	Germany	Montana	None	1545	$ 5.00	$ 12.00	$ 18,540.00	$ -	$ 18,540.00	$ 4,635.00	$ 13,905.00	01/06/2014	6	June	2014
12	Midmarket	Mexico	Montana	None	2470	$ 5.00	$ 15.00	$ 37,050.00	$ -	$ 37,050.00	$ 24,700.00	$ 12,350.00	01/06/2014	6	June	2014
13	Enterprise	Canada	Montana	None	2665.5	$ 5.00	$ 125.00	$ 333,187.50	$ -	$ 333,187.50	$ 319,860.00	$ 13,327.50	01/07/2014	7	July	2014
14	Small Business	Mexico	Montana	None	958	$ 5.00	$ 300.00	$ 287,400.00	$ -	$ 287,400.00	$ 239,500.00	$ 47,900.00	01/08/2014	8	August	2014
15	Government	Germany	Montana	None	2146	$ 5.00	$ 7.00	$ 15,022.00	$ -	$ 15,022.00	$ 10,730.00	$ 4,292.00	01/09/2014	9	September	2014
16	Enterprise	Canada	Montana	None	345	$ 5.00	$ 125.00	$ 43,125.00	$ -	$ 43,125.00	$ 41,400.00	$ 1,725.00	01/10/2013	10	October	2013
17	Midmarket	United States of America	Montana	None	615	$ 5.00	$ 15.00	$ 9,225.00	$ -	$ 9,225.00	$ 6,150.00	$ 3,075.00	01/12/2014	12	December	2014
18	Government	Canada	Paseo	None	292	$ 10.00	$ 20.00	$ 5,840.00	$ -	$ 5,840.00	$ 2,920.00	$ 2,920.00	01/02/2014	2	February	2014
19	Midmarket	Mexico	Paseo	None	974	$ 10.00	$ 15.00	$ 14,610.00	$ -	$ 14,610.00	$ 9,740.00	$ 4,870.00	01/02/2014	2	February	2014
20	Channel Partners	Canada	Paseo	None	2518	$ 10.00	$ 12.00	$ 30,216.00	$ -	$ 30,216.00	$ 7,554.00	$ 22,662.00	01/06/2014	6	June	2014
21	Government	Germany	Paseo	None	1006	$ 10.00	$ 350.00	$ 352,100.00	$ -	$ 352,100.00	$ 261,560.00	$ 90,540.00	01/06/2014	6	June	2014
22	Channel Partners	Germany	Paseo	None	367	$ 10.00	$ 12.00	$ 4,404.00	$ -	$ 4,404.00	$ 1,101.00	$ 3,303.00	01/07/2014	7	July	2014
23	Government	Mexico	Paseo	None	883	$ 10.00	$ 7.00	$ 6,181.00	$ -	$ 6,181.00	$ 4,415.00	$ 1,766.00	01/08/2014	8	August	2014
24	Midmarket	France	Paseo	None	549	$ 10.00	$ 15.00	$ 8,235.00	$ -	$ 8,235.00	$ 5,490.00	$ 2,745.00	01/09/2013	9	September	2013
25	Small Business	Mexico	Paseo	None	788	$ 10.00	$ 300.00	$ 236,400.00	$ -	$ 236,400.00	$ 197,000.00	$ 39,400.00	01/09/2013	9	September	2013
26	Midmarket	Mexico	Paseo	None	2472	$ 10.00	$ 15.00	$ 37,080.00	$ -	$ 37,080.00	$ 24,720.00	$ 12,360.00	01/09/2014	9	September	2014
27	Government	United States of America	Paseo	None	1143	$ 10.00	$ 7.00	$ 8,001.00	$ -	$ 8,001.00	$ 5,715.00	$ 2,286.00	01/10/2014	10	October	2014
28	Government	Canada	Paseo	None	1725	$ 10.00	$ 350.00	$ 603,750.00	$ -	$ 603,750.00	$ 448,500.00	$ 155,250.00	01/11/2013	11	November	2013
29	Channel Partners	United States of America	Paseo	None	912	$ 10.00	$ 12.00	$ 10,944.00	$ -	$ 10,944.00	$ 2,736.00	$ 8,208.00	01/11/2013	11	November	2013
30	Midmarket	Canada	Paseo	None	2152	$ 10.00	$ 15.00	$ 32,280.00	$ -	$ 32,280.00	$ 21,520.00	$ 10,760.00	01/12/2013	12	December	2013
31	Government	Canada	Paseo	None	1817	$ 10.00	$ 20.00	$ 36,340.00	$ -	$ 36,340.00	$ 18,170.00	$ 18,170.00	01/12/2014	12	December	2014
32	Government	Germany	Paseo	None	1513	$ 10.00	$ 350.00	$ 529,550.00	$ -	$ 529,550.00	$ 393,380.00	$ 136,170.00	01/12/2014	12	December	2014
33	Government	Mexico	Velo	None	1493	$ 120.00	$ 7.00	$ 10,451.00	$ -	$ 10,451.00	$ 7,465.00	$ 2,986.00	01/01/2014	1	January	2014
34	Enterprise	France	Velo	None	1804	$ 120.00	$ 125.00	$ 225,500.00	$ -	$ 225,500.00	$ 216,480.00	$ 9,020.00	01/02/2014	2	February	2014
35	Channel Partners	Germany	Velo	None	2161	$ 120.00	$ 12.00	$ 25,932.00	$ -	$ 25,932.00	$ 6,483.00	$ 19,449.00	01/03/2014	3	March	2014
36	Government	Germany	Velo	None	1006	$ 120.00	$ 350.00	$ 352,100.00	$ -	$ 352,100.00	$ 261,560.00	$ 90,540.00	01/06/2014	6	June	2014
37	Channel Partners	Germany	Velo	None	1545	$ 120.00	$ 12.00	$ 18,540.00	$ -	$ 18,540.00	$ 4,635.00	$ 13,905.00	01/06/2014	6	June	2014
38	Enterprise	United States of America	Velo	None	2821	$ 120.00	$ 125.00	$ 352,625.00	$ -	$ 352,625.00	$ 338,520.00	$ 14,105.00	01/08/2014	8	August	2014
39	Enterprise	Canada	Velo	None	345	$ 120.00	$ 125.00	$ 43,125.00	$ -	$ 43,125.00	$ 41,400.00	$ 1,725.00	01/10/2013	10	October	2013
40	Small Business	Canada	VTT	None	2001	$ 250.00	$ 300.00	$ 600,300.00	$ -	$ 600,300.00	$ 500,250.00	$ 100,050.00	01/02/2014	2	February	2014
41	Channel Partners	Germany	VTT	None	2838	$ 250.00	$ 12.00	$ 34,056.00	$ -	$ 34,056.00	$ 8,514.00	$ 25,542.00	01/04/2014	4	April	2014
42	Midmarket	France	VTT	None	2178	$ 250.00	$ 15.00	$ 32,670.00	$ -	$ 32,670.00	$ 21,780.00	$ 10,890.00	01/06/2014	6	June	2014
43	Midmarket	Germany	VTT	None	888	$ 250.00	$ 15.00	$ 13,320.00	$ -	$ 13,320.00	$ 8,880.00	$ 4,440.00	01/06/2014	6	June	2014
44	Government	France	VTT	None	1527	$ 250.00	$ 350.00	$ 534,450.00	$ -	$ 534,450.00	$ 397,020.00	$ 137,430.00	01/09/2013	9	September	2013
45	Small Business	France	VTT	None	2151	$ 250.00	$ 300.00	$ 645,300.00	$ -	$ 645,300.00	$ 537,750.00	$ 107,550.00	01/09/2014	9	September	2014
46	Government	Canada	VTT	None	1817	$ 250.00	$ 20.00	$ 36,340.00	$ -	$ 36,340.00	$ 18,170.00	$ 18,170.00	01/12/2014	12	December	2014
47	Government	France	Amarilla	None	2750	$ 260.00	$ 350.00	$ 962,500.00	$ -	$ 962,500.00	$ 715,000.00	$ 247,500.00	01/02/2014	2	February	2014
48	Channel Partners	United States of America	Amarilla	None	1953	$ 260.00	$ 12.00	$ 23,436.00	$ -	$ 23,436.00	$ 5,859.00	$ 17,577.00	01/04/2014	4	April	2014
49	Enterprise	Germany	Amarilla	None	4219.5	$ 260.00	$ 125.00	$ 527,437.50	$ -	$ 527,437.50	$ 506,340.00	$ 21,097.50	01/04/2014	4	April	2014

Sheet1 +

Ready 97%

Alternatives

Alternatives

Pie chart

Alternatives

Pie chart

Line graph

Alternatives

Pie chart

Line graph

Scatter plot + regression

Alternatives

Pie chart

Line graph

Scatter plot + regression

Box-and-whisker plot

Alternatives

Pie chart

Line graph

Scatter plot + regression

Box-and-whisker plot

Bar graph

Alternatives

Pie chart

Line graph

Scatter plot + regression

Box-and-whisker plot

Bar graph

Written report

Alternatives

Pie chart

Line graph

Scatter plot + regression

Box-and-whisker plot

Bar graph

Written report

Finger painting

Alternatives

Pie chart

Line graph

Scatter plot + regression

Box-and-whisker plot

Bar graph

Written report

Finger painting

Interpretive dance

Alternatives

Pie chart

Line graph

Scatter plot + regression

Box-and-whisker plot

Bar graph

Written report

Finger painting

Interpretive dance

Smoke signals

Alternatives

Pie chart

Line graph

Scatter plot + regression

Box-and-whisker plot

Bar graph

Written report

Finger painting

Interpretive dance

Smoke signals

Pan flute pop-rock album

Why change?

Why change?

Information hidden/absent/difficult to extract

Why change?

Information hidden/absent/difficult to extract

Representations encode information

Why change?

Information hidden/absent/difficult to extract

Representations encode information

Humans decode information

Why change?

Information hidden/absent/difficult to extract

Representations encode information

Humans decode information

Some information is more important than others

Useful representations

Useful representations

Make important information easy to access

Useful representations

Make important information easy to access

Remove or de-emphasise unimportant information

Useful representations

Make important information easy to access

Remove or de-emphasise unimportant information

Simplify information decoding

Useful representations

Make important information easy to access

Remove or de-emphasise unimportant information

Simplify information decoding

Provide tools to infer new information

Useful representations

Useful representations

Depends on the problem

Useful representations

Depends on the problem

Depends on the solver

Useful representations

Depends on the problem

Depends on the solver

Depends on the task

Useful representations

Depends on the problem

Depends on the solver

Depends on the task

Suggesting options

Suggesting options

Use the answer...

Suggesting options

Use the answer… but that means we’ve solved the problem already!

Suggesting options

Use the answer… but that means we've solved the problem already!

See what the problem looks like in the new representation…

Suggesting options

Use the answer… but that means we've solved the problem already!

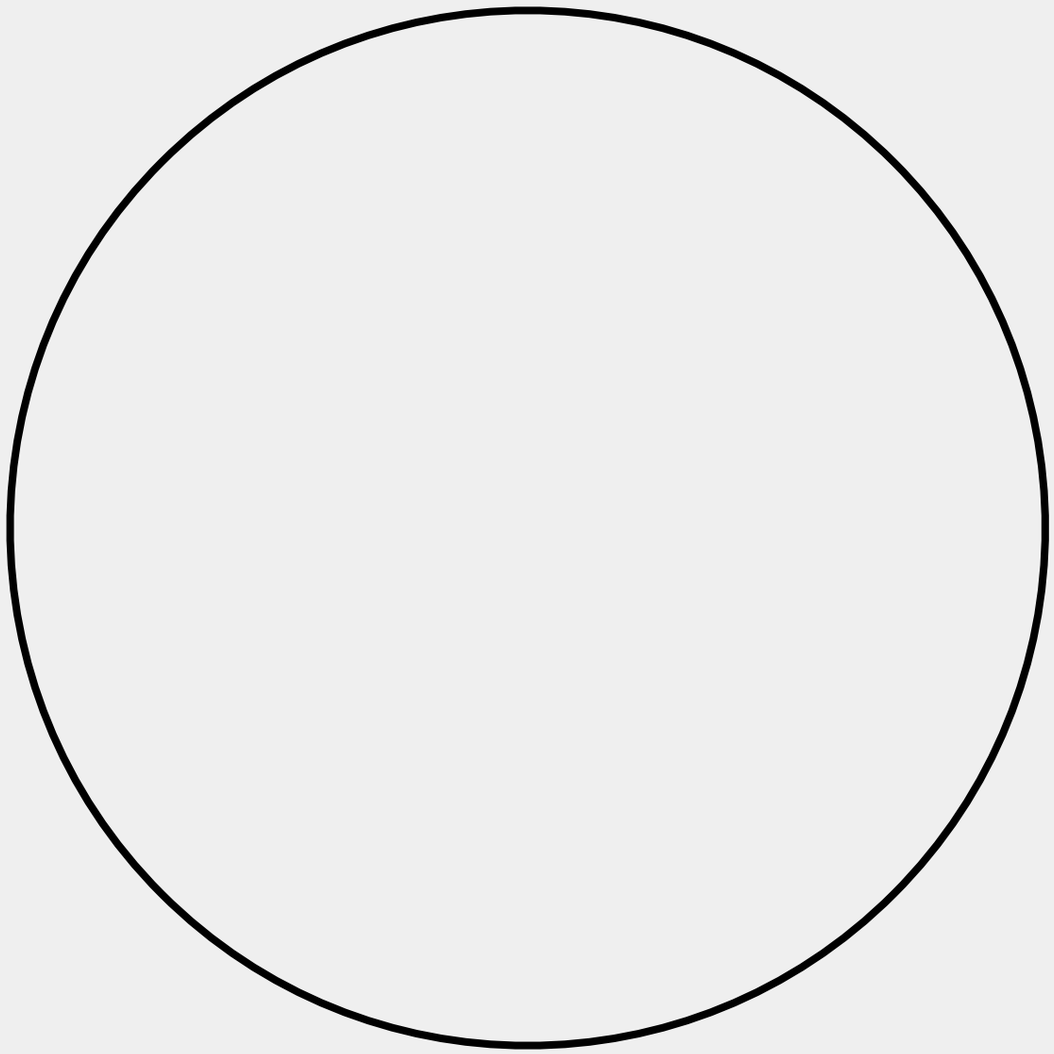
See what the problem looks like in the new representation… but "automatic" translations rarely exist!

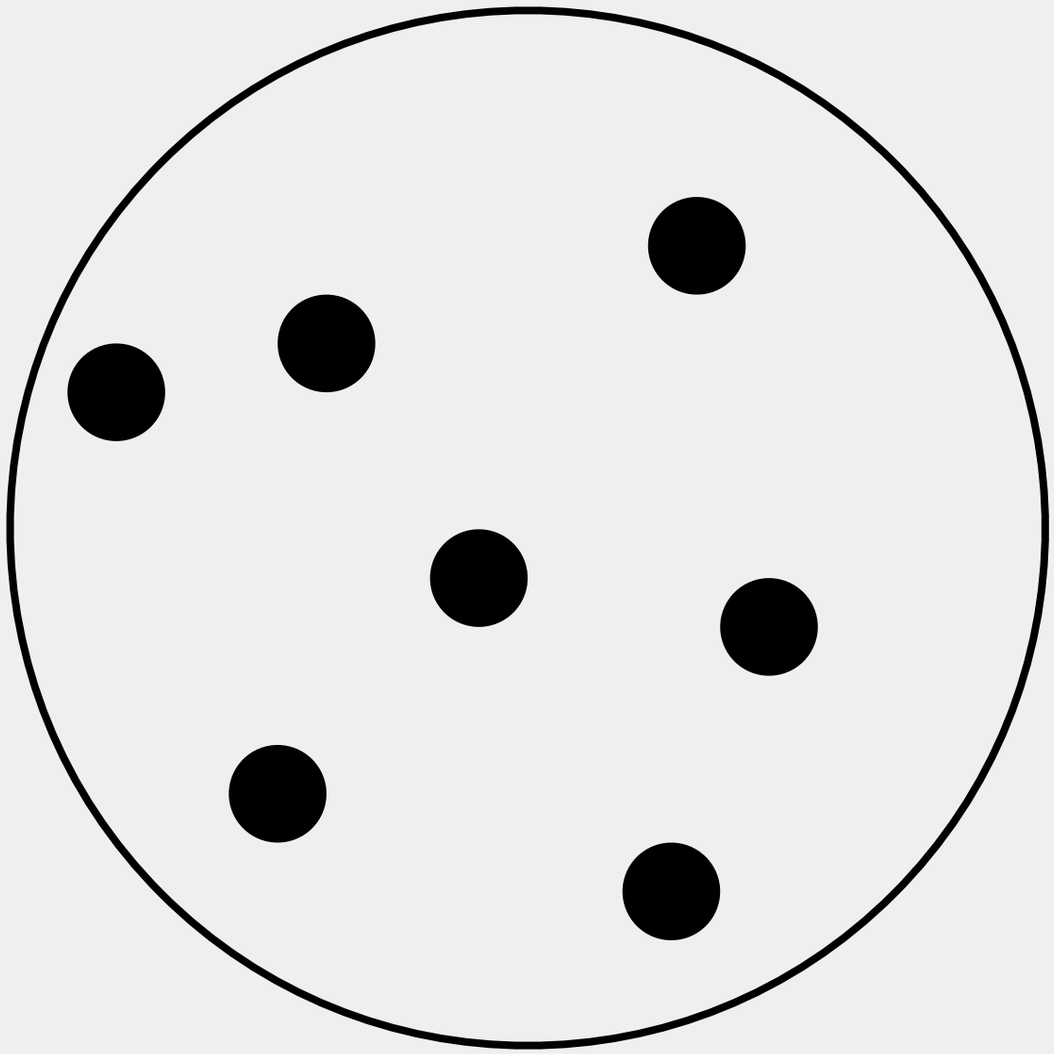
Suggesting options

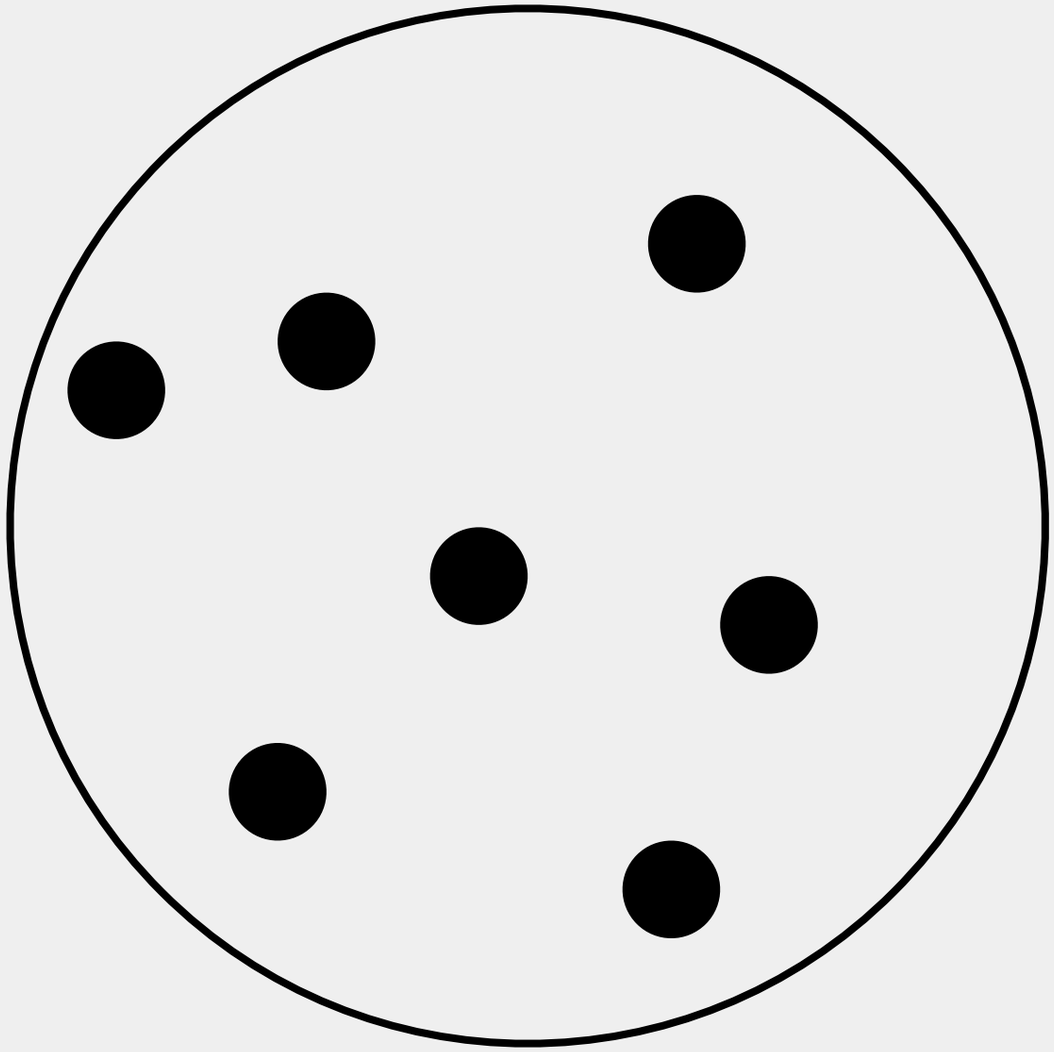
Use the answer… but that means we've solved the problem already!

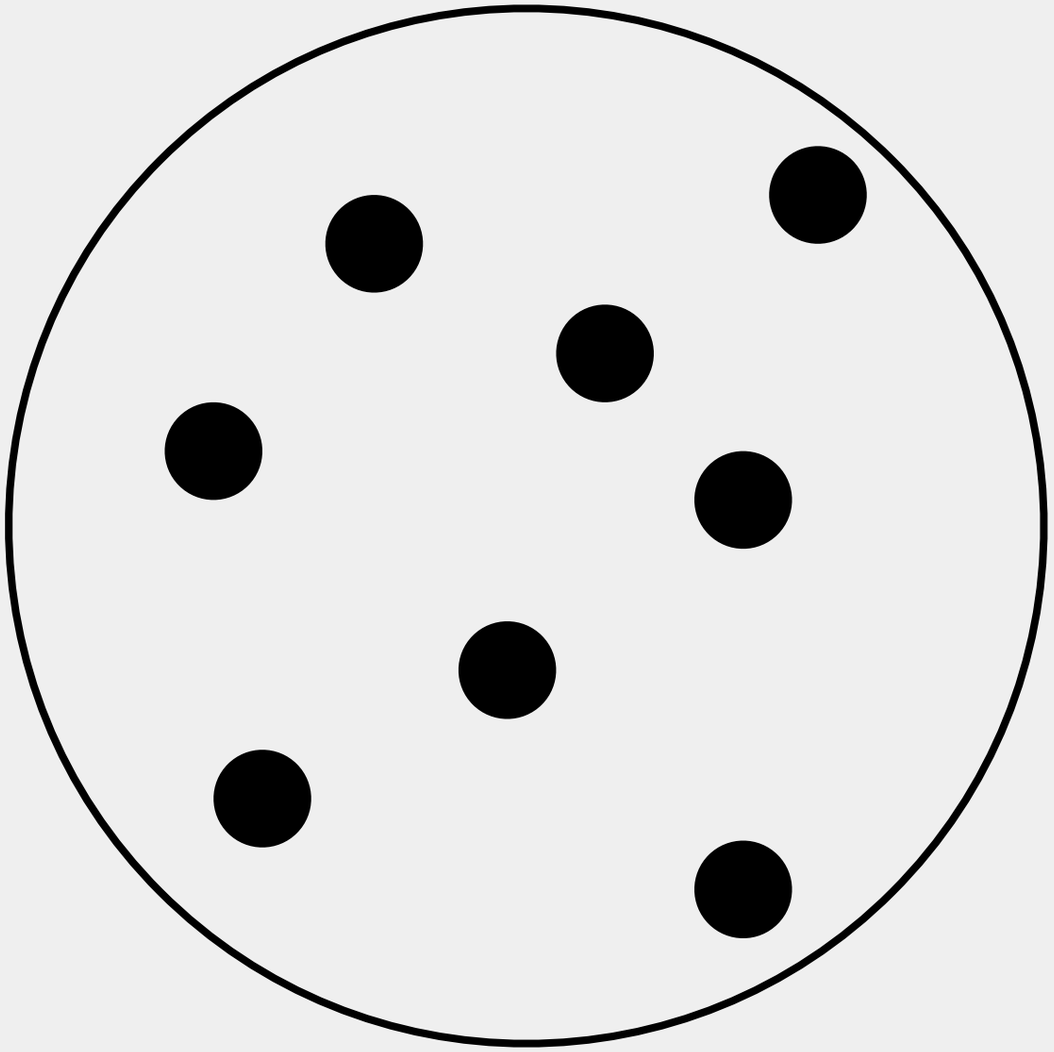
See what the problem looks like in the new representation… but "automatic" translations rarely exist!

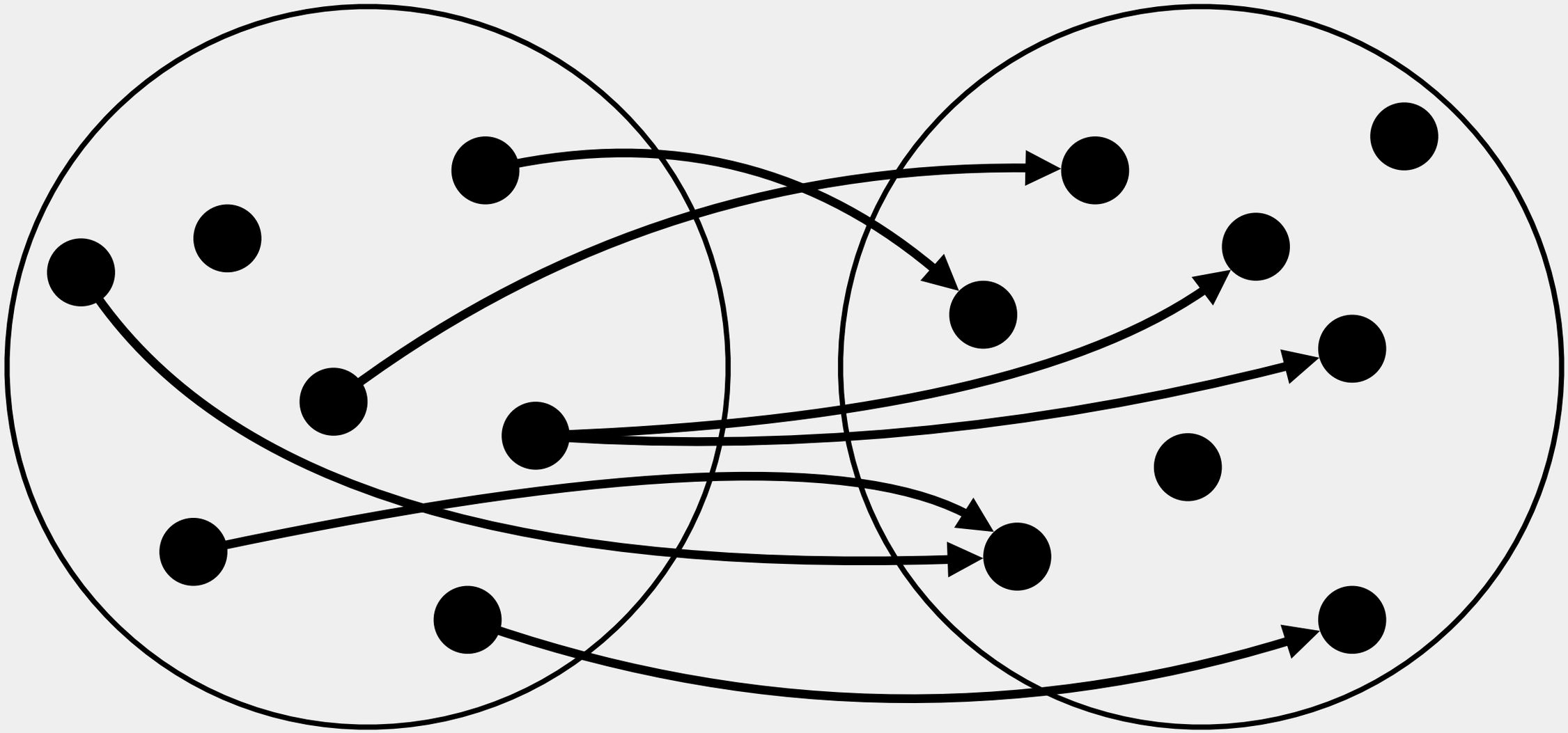
Use relations between properties to to predict what the problem might look like.

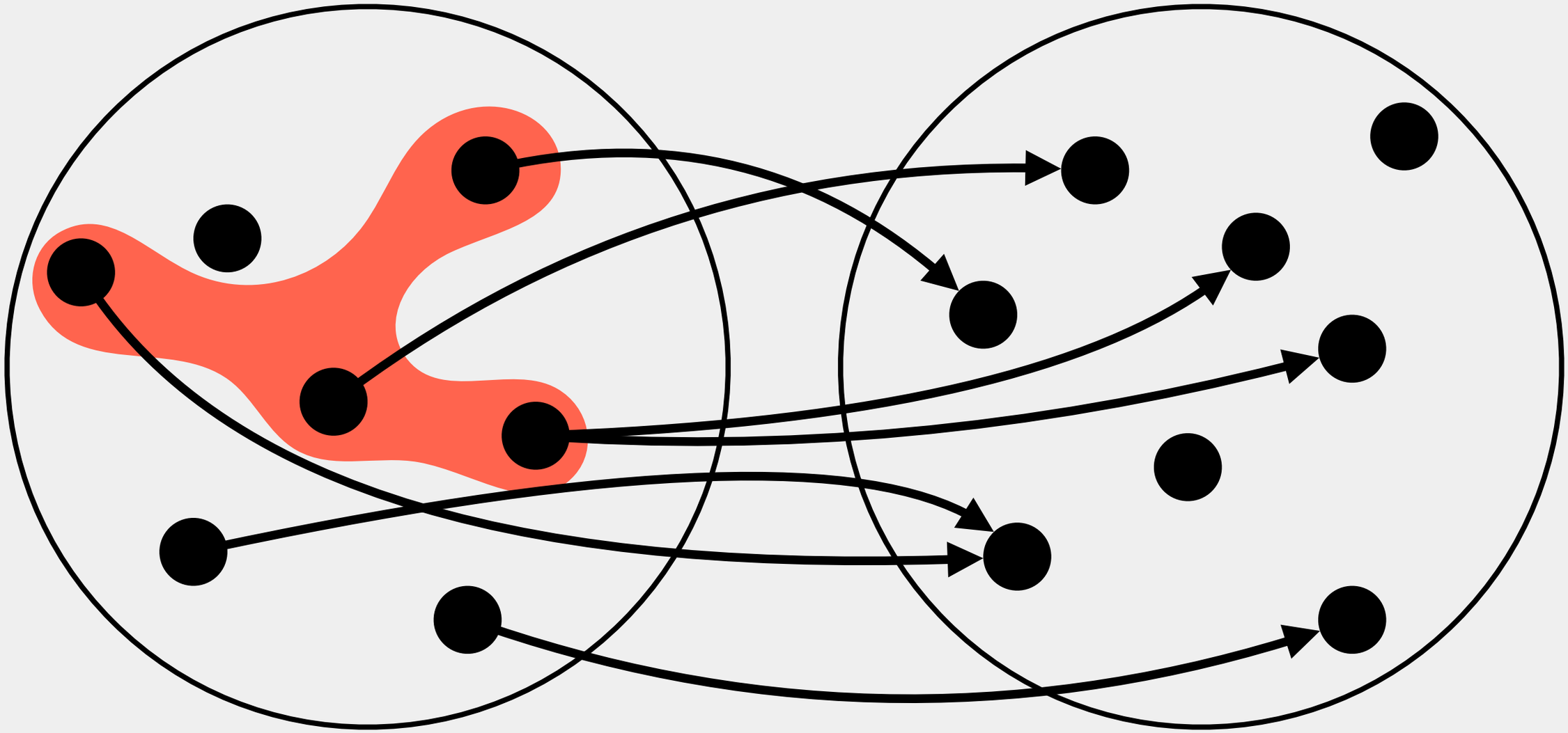


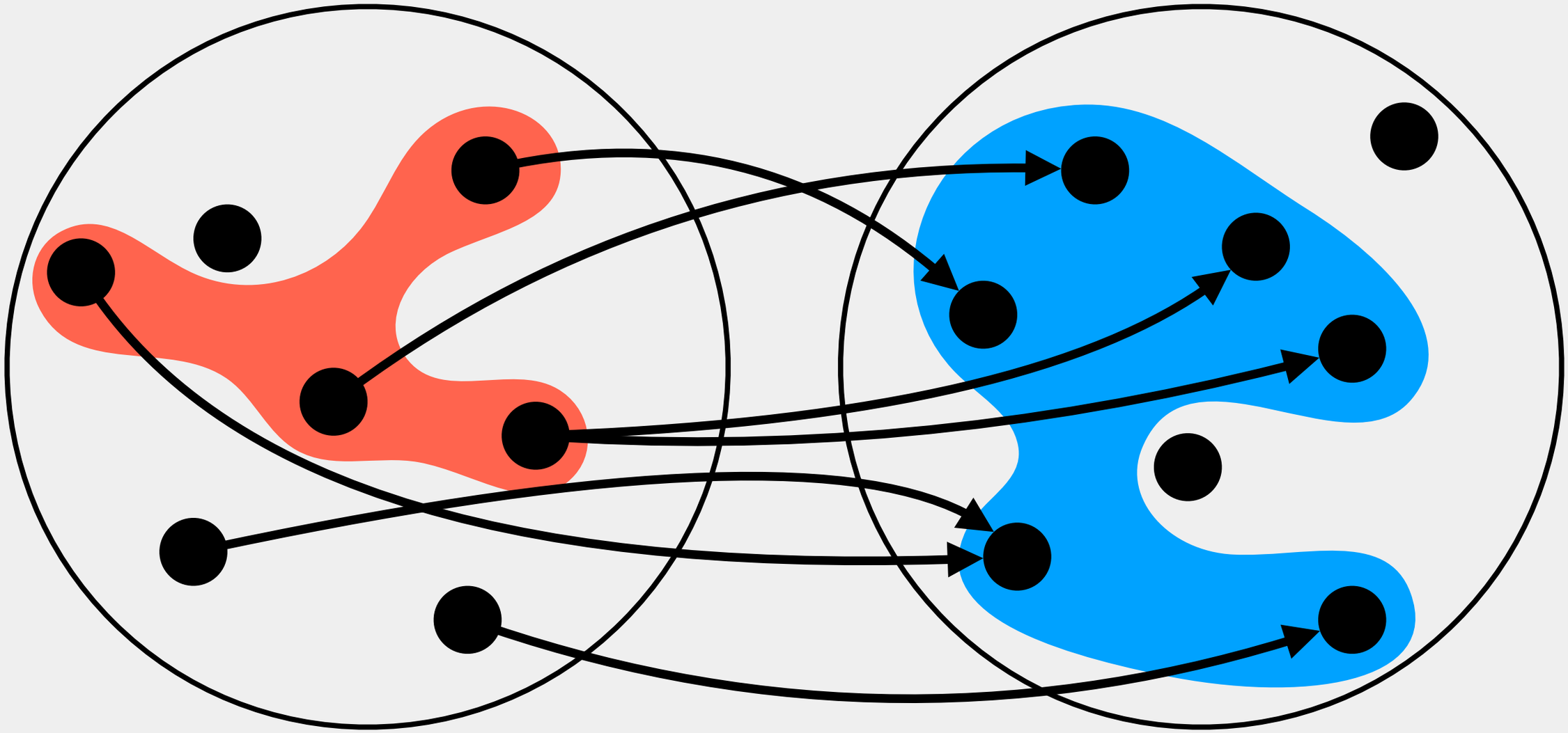


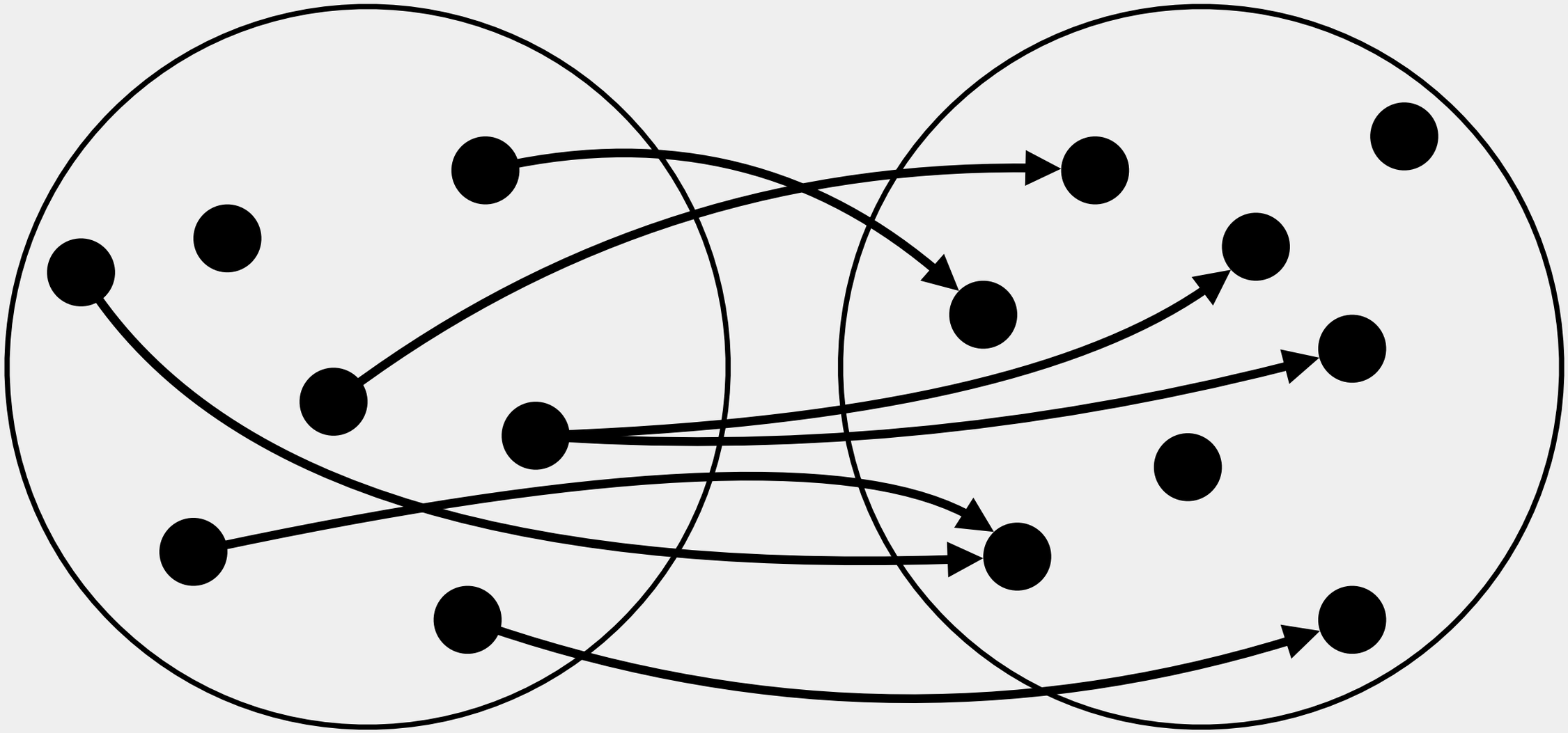












```
import "util.set";
import "util.type";
import "util.dictionary";

import "strategies.property_importance";

signature PROPERTY =
sig
    type property;

    exception ParseError;

    val compare : property * property -> order;
    val propertyMatch : property * property -> bool;
    val nameOf : property -> string;
    val toString : property -> string;
    val fromString : string -> property;
end;

structure Property :> PROPERTY =
struct

datatype property = Simple of string
                  | Typed of (string * Type.T)
                  | Attr of (string * string list);

exception ParseError;

fun nameOf (Simple s) = s
  | nameOf (Typed (s,_)) = s
  | nameOf (Attr (s,_)) = s;

(*A lexicographic order for the property type. The name takes precedence, then
  the KIND (Simple, Typed, Attr), and in the end the type or attribute list.
  Useful for putting it into a dictionary; not for much else*)
fun compare (Simple s, Simple s') = String.compare (s, s')
  | compare (Typed (s,t), Typed (s',t')) = let val c = String.compare (s, s')
                                           in if c = EQUAL
                                              then Type.compare (t,t')
                                              else c
                                           end
  | compare (Attr (s,a), Attr (s',a')) = let val c = String.compare (s, s')
                                         in if c = EQUAL
                                            then List.collate String.compare (a,a')
                                            else c
                                         end
  | compare (p,p') = let val c = String.compare (nameOf p, nameOf p')
                     in if c = EQUAL
                        then case (p,p') of (Simple _, _) => LESS
                                          | (_, Simple _) => GREATER
                                          | (Typed _, _) => LESS
                                          | (_, Typed _) => GREATER
                                          | _ => raise Match
                        else c
                     end  ;

(*propertyMatch is meant to be used for finding whether a correspondence holds
  without the need to have type or attribute information, and type-matching when
  there is type information *)
fun propertyMatch (Simple s, p) = (s = nameOf p)
  | propertyMatch (p, Simple s) = (s = nameOf p)
  | propertyMatch (Typed (s,t), Typed (s',t')) = (s = s' andalso Type.match t t')
  | propertyMatch (Attr (s,_), Attr (s',_)) = (s = s')
  | propertyMatch _ _ = false

fun toString (Simple s) = s
  | toString (Typed (s,t)) = s ^ " : " ^ (Type.toString t)
  | toString (Attr (s,a)) = s ^ " : [" ^ (String.concat (intersperse ", " a)) ^ "]";

(*as-is, fromString is an ugly function. Very ad-hoc.*)
fun fromString x =
  if String.isPrefix "pattern-" x then Simple x else
      case map stringTrim (String.tokens (fn c => c = #":") x) of
          [z,s] =>
              if String.substring (s,0,1) = "[" then
                  if s = "[]" then Attr (z,[])
                  else let
                           fun dropEnds [] = []
                             | dropEnds [x] = []
                             | dropEnds [x, y] = []
                             | dropEnds (x::xs) = List.rev (List.tl (List.rev xs));
                       val s' = String.implode (dropEnds (String.explode s));
                   in
                       Attr (z,map stringTrim (String.tokens (fn c => c = #",") s'))
                   end
              else Type (z, Type.fromString s)
        | [z] => Simple z
        | _ => raise Match;

end;

signature QPROPERTY =
sig
    type property;

    exception ParseError;

    val compare : property * property -> order;
    val toString : property -> string;
    val fromString : string -> property;
    val toPair : property -> (Property.property * Importance.importance);
    val fromPair : (Property.property * Importance.importance) -> property;
    val withoutImportance : property -> Property.property;
end;


structure QProperty :> QPROPERTY =
struct

type property = (Property.property * Importance.importance);
exception ParseError;

val compare = cmpJoin Property.compare Importance.compare;
fun toString (p, i) = "(" ^ (Property.toString p) ^ ", " ^ (Importance.toString i) ^ ")";
fun fromString s =
    let
        val splitter = fn c => c = #"(" orelse  c = #")" orelse c = #",";
    in
        case String.tokens splitter s of
            [a, b] => (case Importance.fromString b of
                           SOME b' => (Property.fromString a, b')
                         | NONE => raise ParseError)
          | _ => raise ParseError
    end;
fun toPair (s, i) = (s, i);
fun fromPair (s, i) = (s, i);
fun withoutImportance (s, _) = s;

end;

structure PropertySet = Set(struct
                            type t = Property.property;
                            val compare = Property.compare;
                            val fmt = Property.toString;
                            end);
structure PropertyDictionary = Dictionary(struct
                                          type k = Property.property;
                                          val compare = Property.compare;
                                          val fmt = Property.toString;
                                          end);
structure QPropertySet = Set(struct
                             type t = QProperty.property;
                             val compare = QProperty.compare;
                             val fmt = QProperty.toString;
                             end);

import "util.set";
import "util.dictionary";

import "strategies.property";
import "strategies.property_importance";

signature CORRESPONDENCE =
sig

    structure S: SET;
    structure D: DICTIONARY;

    exception ParseError

    datatype 'a corrformula = Atom of 'a
                            | Neg of 'a corrformula
                            | Conj of 'a corrformula * 'a corrformula
                            | Disj of 'a corrformula * 'a corrformula;
    type correspondence = Property.property corrformula * Property.property corrformula *
real;

    val equal : correspondence -> correspondence -> bool;
    val match : S.t S.set -> S.t S.set -> correspondence -> bool;
    val matchingIntersectionLeft : S.t S.set -> S.t S.set -> S.t S.set;
    val sameProperties : correspondence -> correspondence -> bool;
    val matchingProperties : correspondence -> correspondence -> bool;
    val strength : correspondence -> real;

    val toString : correspondence -> string;
    val fromString : (string -> Property.property) -> string -> correspondence;
end;

structure Correspondence : CORRESPONDENCE =
struct

structure S = PropertySet;

structure D = PropertyDictionary;

exception ParseError;

datatype 'a corrformula = Atom of 'a
                        | Neg of 'a corrformula
                        | Conj of 'a corrformula * 'a corrformula
                        | Disj of 'a corrformula * 'a corrformula;
type correspondence = Property.property corrformula * Property.property corrformula * real;
```

```
end;

fun fromString propMaker s =
    let
        val temp = (Atom (propMaker s), Atom (propMaker s), 1.0);
        val parts = String.tokens (fn c => c = #",") s;
        val [leftString, rightString, valString] = parts
                                                   handle Match => raise Match;

        fun tokenize string =
            let
                fun cluster [] xs = cluster [[]] xs
                  | cluster cs [] = cs
                  | cluster (c::cs) (x::xs) =
                    case x of
                        #"(" => cluster ([]::[#"("]::c::cs) xs
                      | #")" => cluster ([]::[#")"]::c::cs) xs
                      | s => if Char.isSpace s
                             then cluster ([]::c::cs) xs
                             else cluster ((s::c)::cs) xs;
            in
                List.rev (map (String.implode o List.rev)
                              (List.filter
                                   (fn cs => not (List.null xs))
                                   (cluster [[]] (String.explode string))))
            end;

        fun nextToken [] = NONE
          | nextToken (x::xs) = SOME x;
        fun remainingTokens [] = []
          | remainingTokens (x::xs) = xs;

        fun expect s [] = raise ParseError
          | expect s (x::xs) = if String.toLower(s) = String.toLower(x)
                               then (s, xs)
                               else raise ParseError;

        fun parseAtom [] = raise ParseError
          | parseAtom (x::xs) = if x = "("
                                then raise ParseError
                                else if x = ")"
                                then raise ParseError
                                else if String.toLower(x) = "and"
                                then raise ParseError
                                else if String.toLower(x) = "or"
                                then raise ParseError
                                else if String.toLower(x) = "not"
                                then raise ParseError
                                else (x, xs)
        and parseNeg [] = raise ParseError
          | parseNeg xs =
            let
                val (neg, resta) = expect "not" xs;
                val (atom, restb) = parseCForm resta;
            in
                (Neg atom, restb)
            end
        and parseConj [] = raise ParseError
          | parseConj xs =
            let
                val (left, resta) = parseBForm xs;
                val (andtok, restb) = expect "and" resta;
                val (right, restc) = parseAForm restb;
            in
                (Conj (left, right), restc)
            end
        and parseDisj [] = raise ParseError
          | parseDisj xs =
            let
                val (left, resta) = parseAForm xs;
                val (ortok, restb) = expect "or" resta;
                val (right, restc) = parseFormula restb;
            in
                (Disj (left, right), restc)
            end
        and parseFormula xs = parseDisj xs
                              handle ParseError => parseAForm xs
        and parseAForm xs = parseConj xs
                            handle ParseError => parseBForm xs
        and parseBForm xs = parseNeg xs
                            handle ParseError => parseCForm xs
        and parseCForm xs = parseAtom xs
                            handle ParseError =>
                                   let
                                       val (lparn, resta) = expect "(" xs;
                                       val (toks, restb) = toCloseParen resta;
                                   in
                                       case restb of
                                           [] => parseForm toks
                                         | _ => raise ParseError
                                   end;

        fun parse tokens =
            let
                val (tree, tokens) = parseFormula tokens;
            in
```

```
fun formulamap f (Atom a) = Atom (f a)
  | formulamap f (Neg a) = Neg (formulamap f a)
  | formulamap f (Conj (a, b)) = Conj (formulamap f a, formulamap f b)
  | formulamap f (Disj (a, b)) = Disj (formulamap f a, formulamap f b);

fun emptyIntn a b = S.isEmpty (S.intersection a b);

fun strength (_, _, s) = s;

fun filterMatches p ps = S.filter (fn v => Property.propertyMatch (p,v)) ps;

fun propertyMatches p ps = not (S.isEmpty (filterMatches p ps));

fun allPropertiesMatch ps ps' =
    let val contained = S.map (fn p => propertyMatches p ps') ps;
    in all contained end;

fun somePropertiesMatch ps ps' =
    let val contained = S.map (fn p => propertyMatches p ps') ps;
    in any contained end;

fun joinSets [] = S.empty ()
  | joinSets (h::t) = S.union h (joinSets t)

(* finds matches between two property sets,
but only returns the matches of the left (ps) *)
fun matchingIntersectionLeft ps ps' =
    let val x = S.map (fn p => filterMatches p ps) ps'
    in foldr (fn (a,b) => S.union a b) (S.empty ()) x
    end;


fun match qs rs ((qp, qn), (rp, rn), _) =
    (allPropertiesMatch qp qs) andalso
    (allPropertiesMatch rp rs) andalso
    (not (somePropertiesMatch qn qs)) andalso
    (not (somePropertiesMatch rn rs));

fun biMatch (ps,ps') = allPropertiesMatch ps ps' andalso allPropertiesMatch ps' ps

fun matchingProperties ((qp, qn), (rp, rn), _) ((qp', qn'), (rp', rn'), _) =
    biMatch (qp, qp') andalso
    biMatch (qn, qn') andalso
    biMatch (rp, rp') andalso
    biMatch (rn, rn');

fun sameProperties ((qp, qn), (rp, rn), _) ((qp', qn'), (rp', rn'), _) =
    S.equal (qp, qp') andalso
    S.equal (qn, qn') andalso
    S.equal (rp, rp') andalso
    S.equal (rn, rn');

fun equal c c' = sameProperties c c' andalso Real.== ((strength c), (strength c'));

fun toString ((qp, qn), (rp, rn), s) =
    "((" ^
    (S.toString qp) ^
    ", " ^
    (S.toString qn) ^
    "), (" ^
    (S.toString rp) ^
    ", " ^
    (S.toString rn) ^
    ")) -> " ^
    (Real.toString s);

fun normalise (Atom a) = Atom a
  | normalise (Neg a) =
    let
        val a' = normalise a;
    in
        case a' of
            Atom t => Neg (Atom t)
          | Neg t => t
          | Conj (t, u) => normalise (Disj (Neg t, Neg u))
          | Disj (t, u) => normalise (Conj (Neg t, Neg u))
    end
  | normalise (Conj (a, b)) =
    let
        val a' = normalise a;
        val b' = normalise b;
    in
        case (a', b') of
            (Disj (t, u), v) => normalise (Disj (Conj (t, v),
                                                 Conj (u, v)))
          | (t, Disj(u, v)) => normalise (Disj (Conj (t, u),
                                                Disj (Conj (t, v))))
          | (u, v) => Conj (u, v)
    end
  | normalise (Disj (a, b)) =
    let
        val a' = normalise a;
        val b' = normalise b;
    in
        Disj (a', b')
```

```
                    if (List.null tokens) then tree
                    else raise Match
                end;

        val read = (normalise o parse o tokenize);
    in
        (read leftString, read rightString, Real.fromString valString)
    end;

end;
signature IMPORTANCE =
sig

    datatype importance = Noise | Zero | Low | Medium | High;

    val compare : (importance * importance) -> order;
    val fromString : string -> importance option;
    val toString : importance -> string;

end;

structure Importance : IMPORTANCE =
struct

datatype importance = Noise | Zero | Low | Medium | High;

fun compare (a, b) = let
    import fun ordify Noise = -1
      | ordify Zero = 0
      | ordify Low = 1
      | ordify Medium = 2
      | ordify High = 3;
in
    Int.compare (ordify a, ordify b)
end;

fun fromString "Noise" = SOME Noise
  | fromString "Zero" = SOME Zero
  | fromString "Low" = SOME Low
  | fromString "Medium" = SOME Medium
  | fromString "High" = SOME High
  | fromString _ = NONE;

fun toString Noise = "Noise"
  | toString Zero = "Zero"
  | toString Low = "Low"
  | toString Medium = "Medium"
  | toString High = "High";

end;
(*
The following functions handle how properties are generated from the "property"
table. For example, they key "operators" then lists a collection of operators.
To read this, we use the "readCollection" function and will prepend each of the
operators with the string "op-". Compare this with basic labels, which simply
return the one thing that is there, in a list, ready to prepend to. Bools are
simplest, either returning an empty list (false), or a list containing the empty
string (true) to generate either the key, or nothing, as a property.
A concrete example: From the table
  operators       +, -, *, \sqrt
  sentential      true
  logic-power     2
we would generate the properties
  op-+, op--, op-*, op-\sqrt, sentential, logic-power-2
*)

import "strategies.property_importance";
import "strategies.property_tables";

structure PropertyReader =
struct


fun bool str = if ((String.implode (map Char.toLower (String.explode str))) = "true")
               then [""] else [];

fun label str = [str];

fun collection str = if str = "NONE" then []
                     else map stringTrim (String.tokens (fn c => c = #",") str);

fun dimension str =
    let
        fun parseDimProps s = if s = "[]" then []
                              else let
                                  fun dropEnds [] = []
                                    | dropEnds [x] = []
                                    | dropEnds [x, y] = []
                                    | dropEnds (x::xs) = List.rev (List.tl (List.rev xs));
                                  val s' = String.implode (dropEnds (String.explode s));
                              in
                                  map stringTrim (String.tokens (fn c => c = #";") s')
                              end;
        fun createPairs dimval =
            let
                val parts = map stringTrim (String.tokens (fn c => c = #":") dimval);
            in
                case parts of
                    [x, y] => (x, parseDimProps y)
                  | _ => raise PropertyTables.TableError
                             ("Unable to read dimensions from " ^ dimval)
            end;
        val dimensions = collection str;
        val dimensionsWithValues = map createPairs dimensions;
        val dimensionsSplitOut = map (fn (x, y) =>
                                         map (fn z => x ^ "-" ^ z) y)
                                     dimensionsWithValues;
        val dimensionsNoLabels = map (fn (x, y) =>
                                         map (fn z:string => z) y)
                                     dimensionsWithValues;
    in
        List.foldr (fn (a, b) => a @ b) [] (dimensionsSplitOut @ dimensionsNoLabels)
    end;

end;


let
    open Importance;
    fun stripImportance vals = map (fn (l, (f, p, i)) => (l, (f, p))) vals;

    val RSProperties = [
        ("mode",
         (PropertyReader.collection, "mode-")),
        ("grammar-imports",
         (PropertyReader.collection, "import-")),
        ("grammatical-complexity",
         (PropertyReader.label, "grammatical-complexity-")),
        ("rigorous",
         (PropertyReader.bool, "rigorous")),
        ("knowledge-manipulation-system",
         (PropertyReader.bool, "knowledge-manipulation-system")),
        ("facts",
         (PropertyReader.collection, "fact-")),
        ("fact-imports",
         (PropertyReader.collection, "import-")),
        ("tactics",
         (PropertyReader.collection, "tactic-")),
        ("logic-power",
         (PropertyReader.label, "logic-power-")),
        ("standard-accessibility-manipulations",
         (PropertyReader.collection, "accessible-manipulation-")),
        ("accessible-facts",
         (PropertyReader.bool, "accessible-facts")),
        ("accessible-tactics",
         (PropertyReader.bool, "accessible-tactics")),
        ("accessible-grammatical-constructors",
         (PropertyReader.bool, "accessible-grammatical-constructors")),
        ("editable-external-memory",
         (PropertyReader.bool, "editable-external-memory")),
        ("physical-dimension-use",
         (PropertyReader.dimension, "dimension-use-")),
        ("grammatical-dimensionality",
         (PropertyReader.label, "grammatical-dimensionality-")),
        ("grammatical-granularity",
         (PropertyReader.label, "grammatical-granularity-")),
        ("mean-branching-factor",
         (PropertyReader.label, "mean-branching-factor-")),
        ("mean-solution-depth",
         (PropertyReader.label, "mean-solution-depth-")),
        ("pr-distinct-state-change",
         (PropertyReader.label, "pr-distinct-state-change-")),
        ("pr-valid-state-change",
         (PropertyReader.label, "pr-valid-state-change-")),
        ("types",
         (PropertyReader.collection, "type-")),
        ("tokens",
         (PropertyReader.collection, "token-")),
        ("operators",
         (PropertyReader.collection, "token-")),
        ("relations",
         (PropertyReader.collection, "token-")),
        ("patterns",
         (PropertyReader.collection, "pattern-"))
    ];
    val QProperties = [
        ("error-allowed",
         (PropertyReader.label, "error-allowed-", High)),  (*previously rigorous*)
        ("answer-type",
         (PropertyReader.collection, "type-", High)), (*previously question-value-type*)
        ("instrumental-tokens",
         (PropertyReader.collection, "token-", Medium)),
        ("instrumental-types",
         (PropertyReader.collection, "type-", Medium)),
        ("instrumental-patterns",
         (PropertyReader.collection, "pattern-", Medium)),
        ("instrumental-facts",
         (PropertyReader.collection, "fact-", Medium)),
        ("instrumental-tactics",
         (PropertyReader.collection, "tactic-", Medium)),
        ("relevant-tokens",
         (PropertyReader.collection, "token-", Low)),
        ("relevant-related-tokens",
         (PropertyReader.collection, "token-", Low)),
        ("num-statements",
         (PropertyReader.label, "num-statements-", Zero)),
        ("num-tokens",
         (PropertyReader.label, "num-tokens-", Zero)),
        ("num-distinct-tokens",
         (PropertyReader.label, "num-distinct-tokens-", Zero)),
        ("noise-tokens",
         (PropertyReader.collection, "token-", Noise)),
        ("noise-related-tokens",
         (PropertyReader.collection, "token-", Noise))
    ];
    val QandRSProperties = [
    ];
in
    PropertyTables.setQGenerators
        (QProperties @ QandRSProperties);
    PropertyTables.setRSGenerators
        (RSProperties @ (stripImportance QandRSProperties))
end;
import "util.logging";
import "util.set";
import "util.dictionary";
import "util.csv";

import "strategies.property";
import "strategies.property_importance";
import "strategies.property_correspondence";

signature PROPERTYTABLES =
sig
    exception TableError of string;

    structure FileDict : DICTIONARY;

    type qgenerator = (string -> string list) * string * Importance.importance;
    type rsgenerator = (string -> string list) * string;

    val loadCorrespondenceTable : string -> Correspondence.correspondence list;
    val loadQuestionTable : string -> (FileDict.k, QPropertySet.t QPropertySet.set)
FileDict.dict;
    val loadRepresentationTable : string -> (FileDict.k, PropertySet.t PropertySet.set)
FileDict.dict;

    val setQGenerator : (string * qgenerator) -> unit;
    val setQGenerators : (string * qgenerator) list -> unit;
    val setRSGenerator : (string * rsgenerator) -> unit;
    val setRSGenerators : (string * rsgenerator) list -> unit;

end;


structure PropertyTables : PROPERTYTABLES =
struct

exception TableError of string;

structure SQ = QPropertySet;
structure S = PropertySet;
val qset' = SQ.fromList;
val set' = S.fromList;

structure D = PropertyDictionary;
val dict' = D.fromPairList;
fun getValue d k = SOME (D.get d k)
                   handle D.KeyError => NONE;

structure FileDict = Dictionary(struct
                                type k = string;
                                val compare = String.compare;
                                val fmt = (fn s => s);
                                end);
val filedict' = FileDict.fromPairList;

structure GenDict = Dictionary(struct
                               type k = string;
                               val compare = String.compare;
                               val fmt = (fn s => s);
                               end);
val gdict' = GenDict.fromPairList;

structure CSVLiberal = CSVIO(struct val delimiters = [#","];
                                    val newlines = ["\r", "\n", "\r\n"];
                             end);

type qgenerator = (string -> string list) * string * Importance.importance;
type rsgenerator = (string -> string list) * string;

datatype CorrTree = Prop of string
                  | Neg of CorrTree
                  | Conj of CorrTree * CorrTree
                  | Disj of CorrTree * CorrTree;


fun readCorrespondence qpString rspString strengthString =
```

```
let
    fun tokenize string =
        let
            fun cluster [] xs = cluster [[]] xs
              | cluster cs [] = cs
              | cluster (c::cs) (x::xs) =
                case x of
                    #"(" => cluster ([]::[#"("]::c::cs) xs
                  | #")" => cluster ([]::[#")"]::c::cs) xs
                  | s => if Char.isSpace s
                         then cluster ([]::c::cs) xs
                         else cluster ((s::c)::cs) xs;
        in
            List.rev
                (map (String.implode o List.rev)
                     (List.filter
                          (fn cs => not (List.null cs))
                          (cluster [[]] (String.explode string))))
        end;

    fun parseAtom tokens =
        let
            val nextTok = if (List.null tokens) then NONE
                          else (SOME (List.hd tokens));
        in
            case nextTok of
                SOME "(" =>
                let
                    val useTokens = List.tl tokens;
                    val (disjtree, remainingTokens) = parseDisj useTokens;
                    val closePar = if (List.null remainingTokens) then NONE
                                   else (SOME (List.hd remainingTokens));
                in
                    case closePar of
                        SOME ")" => (disjtree, (List.tl remainingTokens))
                      | SOME other => raise TableError "Missing closing paren"
                      | NONE => (disjtree, [])
                end
              | SOME other => (Prop other, (List.tl tokens))
              | NONE => raise TableError "Unexpected EOF"
        end
    and parseNeg tokens =
        let
            val nextTok = if (List.null tokens) then NONE
                          else (SOME (List.hd tokens));
        in
            case nextTok of
                SOME "NOT" =>
                let
                    val useTokens = List.tl tokens;
                    val (atomTree, remainingTokens) = parseAtom useTokens;
                in
                    (Neg atomTree, remainingTokens)
                end
              | SOME other => parseAtom tokens
              | NONE => raise TableError "Unexpected EOF"
        end
    and parseConj tokens =
        let
            val (readIn, nextTokens) = parseNeg tokens;
            val nextTok = if (List.null nextTokens) then NONE
                          else (SOME (List.hd nextTokens));
        in
            case nextTok of
                SOME "AND" =>
                let
                    val useTokens = List.tl nextTokens;
                    val (conjtree, remainingTokens) = parseConj useTokens;
                in
                    (Conj (readIn, conjtree), remainingTokens)
                end
              | SOME other => (readIn, nextTokens)
              | NONE => (readIn, [])
        end
    and parseDisj tokens =
        let
            val (readIn, nextTokens) = parseConj tokens;
            val nextTok = if (List.null nextTokens) then NONE
                          else (SOME (List.hd nextTokens));
        in
            case nextTok of
                SOME "OR" =>
                let
                    val useTokens = List.tl nextTokens;
                    val (distree, remainingTokens) = parseDisj useTokens;
                in
                    (Disj (readIn, distree), remainingTokens)
                end
              | SOME other => (readIn, nextTokens)
              | NONE => (readIn, [])
        end;

    fun parse tokens =
        let
            val (tree, tokens) = parseDisj tokens;
        in
            if (List.null tokens) then tree else raise TableError "Unprocessed content"
        end;

    fun normalise (Prop s) = Prop s
      | normalise (Neg s) =
        let
            val s' = normalise s;
        in
            case s' of
                Prop k => Neg (Prop k)
              | Neg k => normalise k
              | Conj (a, b) => normalise (Disj (Neg a, Neg b))
              | Disj (a, b) => normalise (Conj (Neg a, Neg b))
        end
      | normalise (Conj (a, b)) =
        let
            val a' = normalise a;
            val b' = normalise b;
        in
            case (a', b') of
                (Disj (u, v), w) => normalise (Disj (Conj (u, w),
                                                     Conj (v, w)))
              | (u, Disj (v, w)) => normalise (Disj (Conj (u, v),
                                                     Conj (u, w)))
              | (u, v) => Conj (u, v)
        end
      | normalise (Disj (a, b)) =
        let
            val a' = normalise a;
            val b' = normalise b;
        in
            Disj (a', b')
        end;

    fun setify (Prop s) = (set' [Property.fromString s], S.empty ())
      | setify (Neg (Prop s)) = (S.empty (), set' [Property.fromString s])
      | setify (Neg _) = raise TableError
                               "Correspondences incorrectly normalised"
      | setify (Conj (a, b)) =
        let
            fun conjFlatten (Prop s) = [Prop s]
              | conjFlatten (Neg a) = [Neg a]
              | conjFlatten (Conj (a, b)) = (conjFlatten a) @ (conjFlatten b)
              | conjFlatten (Disj _) = raise TableError
                                             "Correspondences incorrectly normalised";
            fun isPos (Prop s) = true
              | isPos _ = false;
            fun isNeg (Neg a) = true
              | isNeg _ = false;
            fun stripTreeness (Prop s) = Property.fromString s
              | stripTreeness (Neg (Prop s)) = Property.fromString s
              | stripTreeness _ = raise TableError
                                        "Correspondences incorrectly normalised";
            val flattened = ConjFlatten (Conj (a, b));
            val positives = set' (map stripTreeness (List.filter isPos flattened));
            val negatives = set' (map stripTreeness (List.filter isNeg flattened));
        in
            (positives, negatives)
        end
      | setify (Disj _) = raise TableError
                                "Correspondences incorrectly normalised";

    val read = (normalise o parse o tokenize);

    fun strength str =
        case (Real.fromString str) of
            SOME v => v
          | NONE => raise TableError "Correspondence strength is not a float";

    fun genCorr (Disj (a, b)) t2 v =
        let
            val expa = genCorr a t2 v;
            val expb = genCorr b t2 v;
            val expc = genCorr (normalise (Conj (a, b))) t2 (~v);
        in
            expa @ expb @ expc
        end
      | genCorr t1 (Disj (a, b)) v =
        let
            val expa = genCorr t1 a v;
            val expb = genCorr t1 b v;
            val expc = genCorr t1 (normalise (Conj (a, b))) (~v);
        in
            expa @ expb @ expc
        end
      | genCorr t1 t2 v = [(t1, t2, v)];


    val qpTree = read qpString;
    val rspTree = read rspString;
    val corVal = strength strengthString;
in
    map
        (fn (t1, t2, v) => (setify t1, setify t2, v))
        (genCorr qpTree rspTree corVal)
end;

fun loadCorrespondenceTable filename =
    let
        fun makeRow [x, y] = (readCorrespondence x y "1.0")
          | makeRow [x, y, z] = (readCorrespondence x y z)
          | makeRow r = raise TableError
                              ("Correspondence table entry malformed: " ^
                               (listToString (fn s => s) r));
        val csvFile = CSVLiberal.openIn filename;
        val csvData = CSVLiberal.input csvFile;
    in
        List.foldr (fn (r, xs) => (makeRow r) @ xs) [] csvData
    end
    handle IO.Io e => (Logging.error ("ERROR: File '" ^ filename ^ "' could not be
loaded\n");
                       raise (IO.Io e))
         | TableError reason => (
           Logging.error ("ERROR: CSV parsing failed in file '" ^ filename ^ "'\n");
           Logging.error ("       " ^ reason ^ "\n");
           raise TableError reason
         );

fun loadQuestionTable filename = let
    val sets = (SQ.empty, SQ.union);
    fun parseImportance s = case Importance.fromString s of
                                SOME i => i
                              | NONE => raise TableError ("Unknown importance '" ^ s ^ "'");
    fun parseRow [x, y] = (x, y, NONE)
      | parseRow [x, y, z] = (x, y, SOME (parseImportance z))
      | parseRow _ = raise TableError "Malformed question property entry";
```

```
```

Why does this matter?

Why does this matter?

Advances artificial meta-cognition

Why does this matter?

Advances artificial meta-cognition

Collaborative problem solving

Why does this matter?

Advances artificial meta-cognition

Collaborative problem solving

Explainable AI solutions

Applications

Applications

Engineering

Applications

Engineering

Medicine

Applications

Engineering

Medicine

Case law

Applications

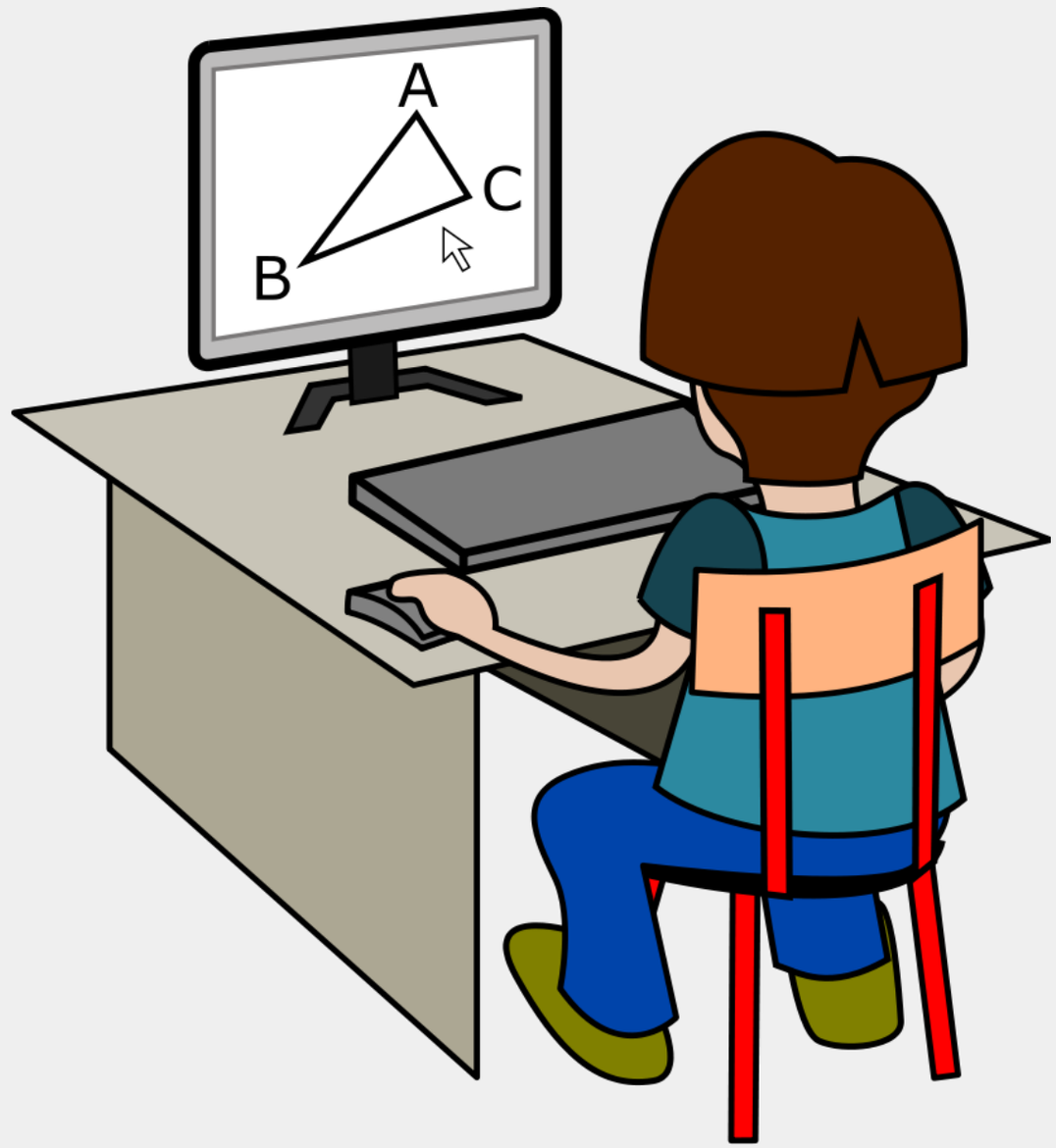
Engineering

Medicine

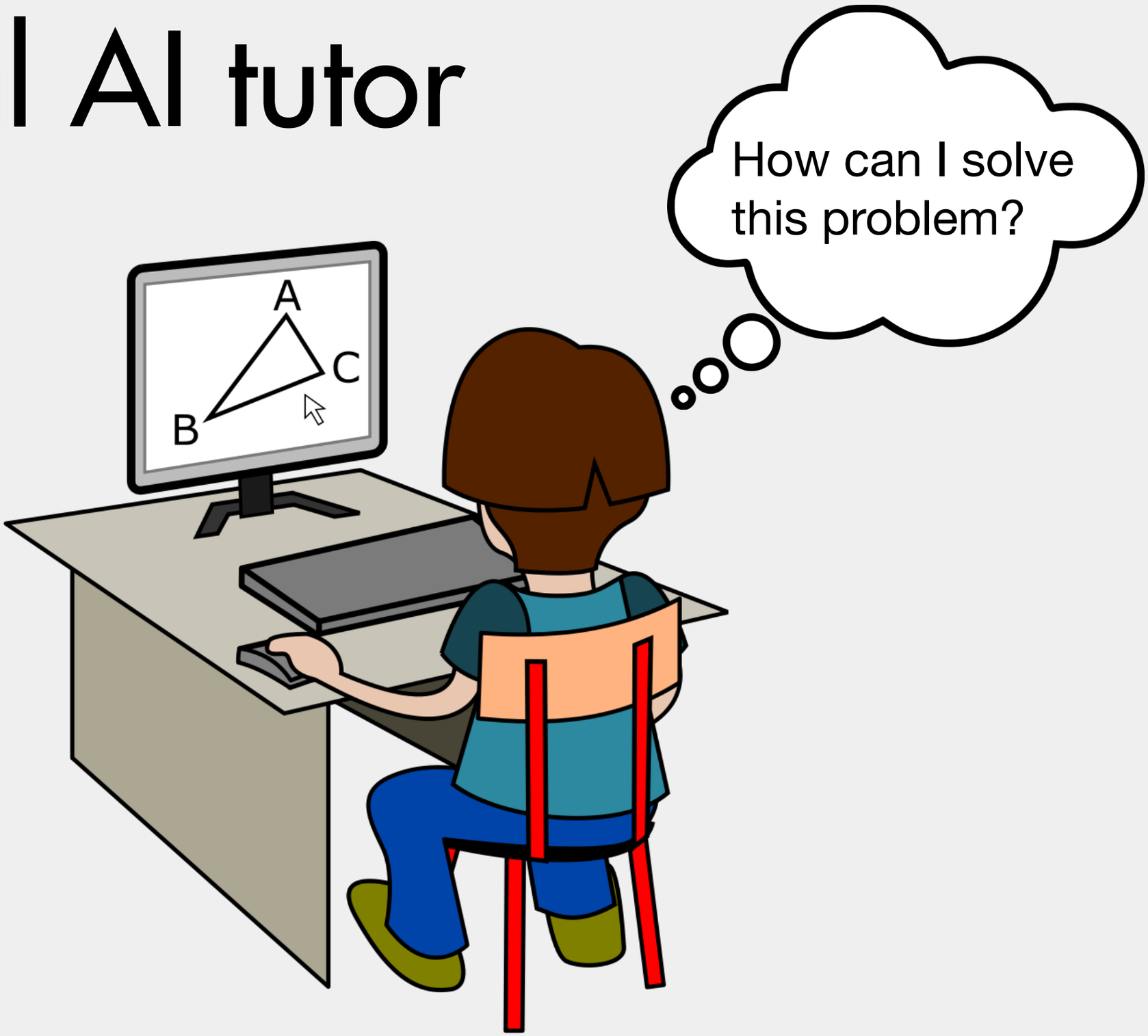
Case law

Education

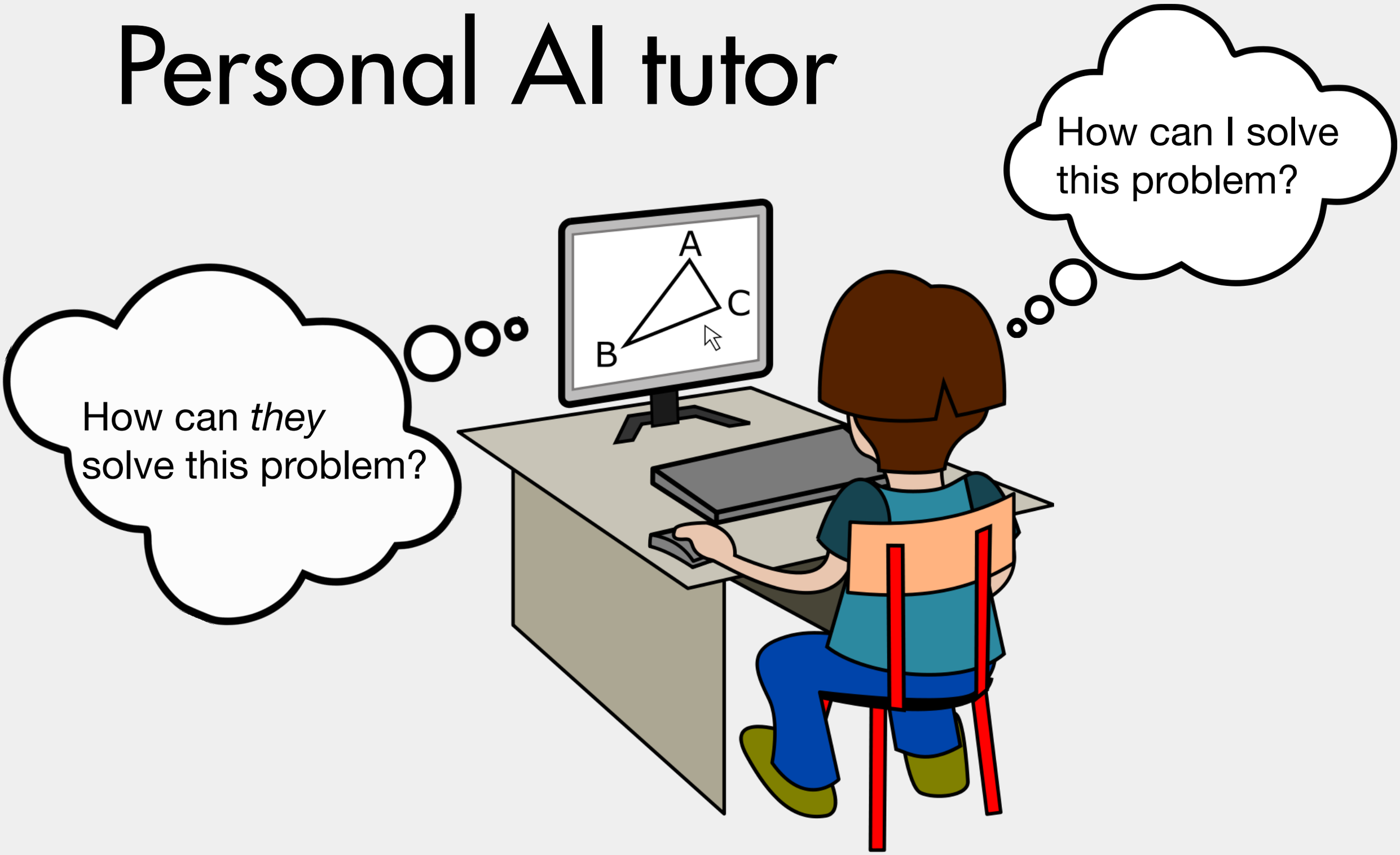
Personal AI tutor



Personal AI tutor



Personal AI tutor



AI is not going to replace people

AI is not going to replace people, it's going to support people

UNIVERSITY OF
CAMBRIDGE
Department of Computer
Science and Technology

Automating representation change across domains for reasoning

Aaron Stockdill
Mateja Jamnik
Daniel Raggi
Peter Cheng
Grecia Garcia Garcia
Holly Sutherland