BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-91-M3

Heuristics for Cost-based Abduction in Belief Networks for Story Understanding

by
Saadia Husain

Heuristics for Cost-based Abduction in in Belief Networks for Story Understanding

by
Saadia Husain
S.B. Massachusetts Institute of Technology, 1989

Project

Submitted in partial fullfilment of the requirements for the
degree of Master of Science in the Department of Computer Science at
Brown University.
May 1991

Eugene Charniak (advisor) 

Heuristics for Cost-based Abduction in Belief Networks for Story Understanding

Masters Project
Saadia Husain

SUMMARY OF PROJECT:

---Cost based abduction vs. probability calculations on belief net
The rule-based system is represented as a weighted and/or DAG which corresponds to a belief network.
Anything that can be done with cost-based abduction on the and/or graph can also be done with probability calculations on the belief net.

---What the belief net looks like: pforms , stms (and/or)
Rules are pforms (probability forms), and-nodes
Belief statements are stms, or-nodes
Cost of assuming a root node is -log(P(node))

---Why old heuristic was too slow: no heuristic, no estimate
The minimal cost proofs were found using a best-first search scheme. The "best" partial expansion at any point in the proof was the one with lowest cost. In the original scheme ,this cost was a measure of the rule costs traversed so far, without any indication of what costs lay ahead.

---What did it explore that was not needed
A path in the expansion could lead up to an assumption node with very high cost which would not be discovered until the last step of the expansion. There was no indicator of that high cost at earlier stages of the partial expansion.

--- Need for a better heuristic
A better heuristic would have some estimate of how expensive it is to expand along a certain rule. This has to be more than just the rule's cost, but also a reflection of the expense of the the root nodes that could be reached by using that rule.

---First tried pushing along single child stms -Not too much improvement.
The first heuristic tried was used in areas where an exact estimate of cost could be pushed down as far as possible. This occurs when a stm has only one relevant child pform (only one rule can expand to that stm). That node can push down its entire cost to its child.
The cost pushed down to a rule is the minimum of the costs its parent nodes are trying to push down. Once you know every node above a certain node has pushed down its exact cost, there is no need to expand above that node when the minimal cost proof is being constructed.

However this heuristic did not lead to a reduction of the large numbers of expanisions since there are very few areas where such pushing down can be carried out. Specifically, it took 2 expansions instead of 4 to assign beliefs for the network constructed after entering "Jack..". But for networks larger than that, there were no major areas where this push down would save expansions.

--- The current sharing heuristic
The current sharing heuristic involves passing down the costs of root nodes. The cost of an expansion is initialized as the estimate obtained after passing down the costs from the assumption nodes, through the rules in network, down to the evidence statements.
Then this estimate is updated as the evidence statements are expanded to other nodes.

--- How costs are passed down
Stms divide their cost equally among their children, in case of the optimistic situation where each of those children would require that stm and would share in bearing its cost.
The cost passed down a pform is the minimum cost of the parent stms.

---Sharing heuristic only looks at pass-down-figure
The relevant costs are the costs of the assumptions and the costs of the rules (the pform and-nodes). After the passing down is initially completed, only the pass-down-figures of the stms and pforms are important. The pass-down-figure of any rule or statement can be seen as an optimistic estimate of the costs that remain above that node (not just the personal cost of that node)

---The cost formula

Cost = Partial-cost + (Pass-down-figure pform)
- (Pass-down-figure stm)* children / Need-figure

When the minimum cost proof is costructed, the estimate is updated each time a rule is traveresed to reach a new stm
Always optimistic estimate.

---Pessimistic cost helps weed out impossible partial candidates
The cost of the minimal cost proof is the sum of the costs of the assumptions necessary for the proof plus the cost of the rules.
Not all assumptions and rules in the net will necessarily be used in the expansion of the evidence nodes. Therefore the worst possible cost for an expansion of the network is equal to the costs of all the assumptions plus the costs of all the rules (pforms).
If at any point, the cost of a partial expansion is greater than this pessimistic estimate, that partial can be discarded from the agenda.
This keeps the maintence of a sorted agenda from being slowed down by partials with extremely high costs.

----Ordering by level
Each partial expansion has a list of stms that have yet to be expanded.
When the sharing heuristic is used, it is necessary to restrict which stms can be expanded next to those whose children stms have all been expanded already. This is enforced by ordering the partial-open list by the level of the stms: Level 1 stms are evidence and all other stms have level = level of highest child + 1.
The lowest level stms appear first in the open list and are expanded first.

---Keeping open sorted, and effect on number of expansions
This constraint on the order in which stms are expanded has affects the number of expansions for the network.

---Stms more than once in no-share open
It is necessary to keep track of how many times a particular stm has been expanded to by some rule. In the best case, all of the stms children would require it, and share the burden of its cost, as anticipated.
In the worst case, only one of the children would have to bear the whole cost, making the cost of that expansion worse than expected.

---Removing duplicates
It is possible to arrive at the same partial expansion in more than one way. To avoid continuing further identical expansions of each of these, we must remove duplicates.

---Advantage of binary tree over heap
The agenda of partial expansions is stored in a binary tree sorted by the partial's cost.
When a new expansion of cost c is inserted in the tree, its cost is

automatically compared with all other expansions of cost c in the agenda. This is where partials with same cost and same open-list, same stm-list can be deleted.

---Cases where non-sharing is better in # of expansions than sharing

. Found case where there are fewer expansions without sharing than with sharing because of the constraints on the open-list
. Examined what was happening with expansions with sharing, to see what wrong branches were being taken
. By chance discovered the case where there can be duplicate partials which lead to redundancies
. Put in code to remove duplicates as they are about to be inserted in to binary tree agenda.
. With this duplicate removing, even more cases had fewer in non-sharing

In the test-suite, the construction that gives this problem is
Jack took a bus to the park.
Fred took a taxi to...

... (more description)..

--Where did we get time improvements and expansion # improvements
**Ok-extension function
Certain rules can only be expanded only if certain statments are not already in the expansions (pform-outs)
Check if
Stm you want to expand is not an out of some rule already used
Rule to be expanded along has no outs already in partial
None of the parents of that pform are outs of some rule already used
Putting in this third check reduces the number of expansions

**dstr-hack

**Agenda was put on binary tree to speed up management of insertion and sorting

** Forget about partial with > pessimistic cost.
THis makes the agenda smaller, and speeds up insertion of new partials into tree

---Timing Results
The time spent is dependent on the number of partial expansions in the agenda.
I used the lisp time function to time the assign-beliefs function and rpg's probability calculations on the net.
Most of it takes less than a second, with or without sharing
Fewer than 50 expansions generally take less than a second with or without sharing, while the prob. calculations for those cases take several seconds (up to 15 sec)

The cases where there are fewer expansions without sharing have a "+" next to the times.

As loaded, the sharing heuristic is turned on .
(the variable *share* is t)
Do (setq *share* nil) to turn it off.

Key:

+ ... Fewer expansions without sharing than with sharing

<--- ... Prob. calcs. on belief net are significantly faster than MAP calc. with sharing.

****

Jack went to the supermarket.
He found some milk on the shelf.
He paid for it.

****

WITH SHARING			WITHOUT SHARING	
#exp	MAP	BNET	#exp	MAP
4	0.67 sec	0.66 seconds	4	0.35 seconds
	1.19	0.64		0.03
8	0.14	3.60	14	0.23
	0.14	3.63		0.11
11	0.19	4.70	34	0.14
	0.20	4.69		0.13
96	4.09	19.43	125	0.55
	2.14	17.82		0.38
8	0.13	3.05	10	0.10
	0.14	3.05		0.09
20	0.22	5.68	22	0.13
	0.25	6.55		0.14
377	2.03	22.48	443	4.22
	2.03	20.29		1.49
570	3.57	24.45	1991	76.61
	7.59	24.42		9.38
91	1.77	8.54	382	5.78
	0.94	6.67		2.43
5	0.10	0.60	5	0.08
	0.11	0.48		0.07
36	0.49	15.82	102	0.55
	0.84	14.73		0.39
6	0.12	1.89	7	0.09
	0.11	2.42		0.08
20	0.70	5.78	22	0.13
	0.22	5.65		0.13
23	0.23	1.55	25	0.14
	0.22	1.54		0.13
199	4.39	33.17	1941	37.73
	2.36	30.51		45.36

****

Bill went to the supermarket.
He paid for some milk.

****

WITH SHARING			WITHOUT SHARING	
#exp	MAP	BNET	#exp	MAP
4	0.10 sec	0.27 sec	4	0.03 sec
	0.10	0.28		0.08
8	0.13	3.16	14	0.12
	0.21	3.66		0.12
11	0.19	4.68	34	0.15
	0.21	4.67		0.15
96	2.15	18.87	125	0.89
	0.79	17.33		0.34
8	0.50	3.05	10	0.11
	0.15	3.05		0.12
20	0.19	5.66	22	0.14
	0.21	5.68		0.12
336	1.88	20.30	443	4.23
	3.25	20.34		1.53
53	0.86	11.94	92	0.56
	0.51	11.11		0.28
119	1.14	6.95	425	6.13
	1.45	7.14		2.49

****
Jack gave the busdriver a token.
He got off at the supermarket.
****

WITH SHARING #exp	MAP	BNET		WITHOUT SHARING #exp	MAP
4	0.10 sec	0.28 sec		4	0.07 sec
	0.10	0.54			
8	0.13	1.08		9	0.10
	0.14	1.07			
11	0.18	5.06		21	0.12
	0.19	4.96			
2114	42.27	96.20		2405	48.40
	31.21	95.71			
2655	50.80	93.47		2479	47.58
	43.88	99.80			
11060	216.13	172.84	<----	13949	717.59
	208.99	150.94			
16	1.09	6.00		18	0.11
	0.44	5.16			
24	0.30	10.05		33	0.16
	0.27	9.70			
35	0.65	12.01		59	0.29
	1.30	12.82			
75	1.76	13.43		181	0.99
	0.95	13.29			
264	8.64	22.46		1956	26.27
	7.61	24.32			

****
Jack went to the liquor-store.
He found some bourbon on the shelf.
****

WITH SHARING #exp	MAP	BNET	WITHOUT SHARING #exp	MAP
4	0.13 sec	0.70 sec	4	0.08 sec
	0.11	0.70		
8	0.19	3.18	14	0.34
	0.16	3.16		
11	1.24	5.27	34	0.15
	0.20	4.69		
96	3.21	17.58	125	0.45
	0.78	18.97		
8	0.16	3.07	10	0.10
	0.16	3.28		
20	0.23	5.68	22	0.14
	0.36	5.92		
377	2.89	21.42	443	1.45
	2.04	20.18		
575	3.68	27.02	1921	7.73
	7.18	25.47		
5	0.14	1.23	5	0.09
	0.11	0.74		
36	0.53	14.76	102	0.33
	0.50	16.27		

****
Bill gave the busdriver a token.
****

WITH SHARING #exp	MAP	BNET		WITHOUT SHARING #exp	MAP	
4	0.14 sec	0.28 sec		4	0.11 sec	
	0.11	0.30				
8	0.17	1.03		9	0.11	
	0.15	1.09				
11	0.25	5.49		21	0.16	
	0.17	5.94				
2114	40.55	97.99		2405	47.39	
	46.00	96.83				
2655	46.28	93.06		2479	46.69	+
	51.19 1	102.18				
11060	214.58	150.21	<-----	13949	736.86	
	211.91	156.31				

****
Fred robbed the liquor-store.
Fred pointed a gun at the owner.
****

WITH SHARING #exp	MAP	BNET	WITHOUT SHARING #exp	MAP
4	0.04 sec	0.32 sec	4	0.11 sec
	0.64	1.12		
8	0.12	0.73	9	0.10
	0.14	1.08		
8	0.10	0.84	9	0.39
	0.14	1.17		
6	0.12	1.37	7	0.11
	0.12	2.47		
20	0.39	4.07	22	0.15
	0.23	5.72		
36	0.37	5.78	62	0.37
	1.45	8.95		
94	0.63	9.88	176	0.60
	0.75	13.79		
5	0.10	0.36	5	0.09
	0.11	0.49		
580	9.97	24.33	1658	14.59
	23.97	34.92		

****
Jack took the bus to the airport.
He bought a ticket.
****

WITH SHARING #exp	MAP	BNET	WITHOUT SHARING #exp	MAP	
4	0.03 sec 0.11	0.20 sec 0.57	4	0.09 sec	
8	0.14 0.16	2.28 3.17	14	0.30	
43	0.23 0.30	5.31 7.61	30	0.20	+
68	0.35 0.54	6.82 9.78	40	0.24	+
135	0.85 0.60	12.42 8.94	59	0.34	+
135	1.30 0.65	9.81 12.42	59	0.42	+
10	0.16 0.17	1.37 1.73	15	0.13	+
652	4.07 5.41	17.88 25.07	1551	24.15	
8	0.13 0.15	2.21 3.10	10	0.12	
20	0.19 0.24	4.09 5.65	22	0.14	
940	5.37 6.03	20.00 28.10	1643	18.82	
2530	21.07 49.85	24.21 32.71	6702	116.21	

****
Bill packed a suitcase.
He went to the airport.
****

WITH SHARING #exp	MAP	BNET	WITHOUT SHARING #exp	MAP
4	0.11 sec 0.11	0.57 sec 1.05	4	0.10 sec
8	0.14 0.14	1.03 1.15	9	0.12
7	0.13 0.13	1.08 1.28	8	0.12
8	0.21 0.15	3.23 3.06	10	0.12
24	0.25 0.26	5.95 5.99	36	0.16
237	1.61 4.10	12.98 14.09	733	3.78
953	6.75 6.49	24.91 29.98	1946	22.86
1909	18.99 20.06	47.85 47.55	8894	295.23

****
Jack went to a restaurant.
He got a milkshake.
****

WITH SHARING #exp	MAP	BNET		WITHOUT SHARING #exp	MAP	
4	0.12 sec	0.27 sec		4	0.12 sec	
	0.11	0.57				
8	0.16	3.17		14	0.16	
	0.15	3.14				
11	0.20	4.69		34	0.17	
	0.17	4.68				
84	0.68	13.94		125	0.45	
	1.70	14.16				
8	0.27	3.15		10	0.12	
	0.15	3.05				
20	0.26	6.08		22	0.16	
	0.25	5.74				
803	4.40	28.82		802	3.50	+
	7.52	29.29				
3812	63.12	38.41	<-----	6206	122.27	
	58.58	38.64				
8837	243.65	29.16	<----			
	233.11	27.10				

****
Bill drank a milkshake with a straw.
****

WITH SHARING #exp	MAP	BNET	WITHOUT SHARING #exp	MAP
4	0.49 sec	0.79 sec	4	0.03 sec
	0.57	1.21		
8	0.22	1.45	9	0.11
	0.13	1.17		
8	0.16	1.17	9	0.11
	2.79	4.13		
24	0.51	13.77	62	0.31
	0.54	8.59		
5	0.156	0.56	5	0.11
	0.12	0.92		
28	4.06	10.27	66	0.30
	0.59	10.46		
24	0.36	10.28	88	0.53
	0.43	10.50		

****
Janet put a straw in a milkshake.
****

WITH SHARING #exp	MAP	BNET	WITHOUT SHARING #exp	MAP
4	0.11 sec	0.55 sec	4	0.04 sec
	0.11	0.29		0.10
8	0.16	1.07	9	0.11
	0.13	1.04		0.20
8	0.16	3.54	14	0.12
	0.15	3.16		0.30
79	0.60	8.07	143	0.52
	0.61	8.04		1.24
5	0.11	0.50	5	0.09
	0.12	0.49		0.10
346	8.00	21.70	1128	10.06
	4.70	19.85		
802	10.34	35.18	2459	21.23
	34.43	37.91		

****
Bill got on a bus.
He got off at a restaurant.
He drank a milkshake with a straw.
****

WITH SHARING #exp	MAP	BNET	WITHOUT SHARING #exp	MAP
4	0.11 sec	0.27 sec	4	0.04 sec
	0.10	0.59		
8	0.14	1.75	9	0.11
	0.13	1.06		
8	0.42	1.49	9	0.09
	0.14	1.19		
419	6.43	31.43	985	8.16
	10.06	35.81		
590	4.21	38.86	1231	9.00
	6.08	36.49		
8	0.16	3.07	10	0.11
	0.16	3.06		
20	0.21	5.68	22	0.17
	0.24	5.66		
5855	77.58	51.98 <----	8869	160.27
	85.08	53.18		
160	3.17	22.85	408	2.43
	2.16	19.98		
301	6.78	13.66	2396	36.16
	5.84	13.11		
6	0.13	1.88	7	0.10
	0.15	1.88		
20	0.25	5.69	22	0.17
	0.23	5.66		
738	8.90	22.62	4918	63.61
	8.58	17.87		
3379	52.15	29.69 <-----	17785	535.85
	58.21	24.64		
247	7.36	10.47	1056	23.09
	5.20	10.18		
5	0.13	0.76	5	0.11
	0.17	0.79		
28	0.36	9.59	66	0.32
	1.43	10.60		
24	0.47	10.65	88	0.43
	0.38	10.22		

****
Jack took a taxi to the park.
****

WITH SHARING			WITHOUT SHARING		
#exp	MAP	BNET	#exp	MAP	
4	0.12 sec	0.90 sec	4	0.03 sec	
	0.13	0.28			
8	0.69	3.88	14	0.12	
	0.17	3.19			
43	0.31	7.60	30	0.17	
	0.35	8.47			
68	0.41	9.40	40	0.20	
	0.96	9.32			
135	0.79	14.12	59	0.89	
	0.68	12.42			
135	1.06	12.56	59	0.28	
	0.68	12.43			
10	0.18	1.99	15	0.10	
	0.19	1.72			
158	1.20	12.46	317	1.45	
	1.61	12.89			

****
Bill took a taxi.
****

WITH SHARING			WITHOUT SHARING		
#exp	MAP	BNET	#exp	MAP	
4	0.11 sec	0.31 sec	4	0.03 sec	
	0.13	0.31			
8	0.23	3.23	14	0.12	
	0.16	3.17			
33	0.29	6.75	20	0.11	+
	0.29	6.73			
51	0.31	6.25	45	0.17	+
	0.33	6.55			

****
Fred took a taxi to the bus-station.
He got on a bus.
****

WITH SHARING				WITHOUT SHARING		
#exp	MAP	BNET		#exp	MAP	
4	0.12 sec	0.27 sec		4	0.05 sec	
	0.13	0.29				
8	0.15	3.16		14	0.13	
	0.17	3.20				
43	0.39	8.63		30	0.18	+
	0.34	7.59				
68	0.89	14.35		40	0.23	+
	0.41	9.35				
135	0.69	12.42		59	0.25	+
	0.68	12.47				
135	0.68	14.29		59	0.28	+
	1.73	12.94				
10	0.95	1.72		15	0.10	
	0.20	1.69				
154	1.32	12.05		327	1.32	
	1.18	12.03				
8	0.16	3.05		10	0.12	
	0.16	3.05				
20	0.24	5.67		22	0.15	
	0.24	5.68				
240	3.43	17.41		349	1.20	
	2.49	16.22				
2682	41.62	38.29	<----	3819	91.19	

```
;; -*-Mode: Lisp;  Package: frail;  Base:10.; Syntax: Common-Lisp -*-
;; Sccsid( @(#)costs.lisp



;;*****************************************************************
;;*****************************************************************

;; Code for pushing down costs in belief net
;;*****************************************************************
;;*****************************************************************


(in-package 'frail)

;#################################################################

(defstruct (partial
             (:print-function
              (lambda (p sm i)
                (declare (ignore i))
                (let (assumed rest)
                  (loop for pf in (partial-pforms p)
                        for s = (pform-child pf)
                        do (if (null (pform-parents pf)) (push s assumed)
                               (push s rest)))
                  (format sm
                "#P[cost: ~S ~% open: ~S ~% stms: ~S ~% assumed: ~S ~%]"
                            (partial-cost p) (partial-open p) (partial-stms p)
                            assumed)))))

  cost open stms pforms outs )

;#################################################################

(defvar *cost-fudge* .5)
(defvar *share* t)        ;; Can be set for sharing or non-sharing
(defvar *pessimistic-cost* 0)


;--------- Additional fields for PFORMS and STMS-----------
(defmacro pform-cost (pf) '(first (pform-unused ,pf)))
(defmacro pform-pass-down-figure (pf) '(second (pform-unused ,pf)))
(defmacro pform-outs (pf) '(third (pform-unused ,pf)))

(defmacro stm-rel-pforms (stm) '(first (stm-unused ,stm)))
(defmacro stm-rel-child-pforms (stm) '(second (stm-unused ,stm)))
(defmacro stm-pass-down-figure (stm) '(third (stm-unused ,stm)))
(defmacro stm-level (stm) '(fourth (stm-unused ,stm)))


(defun assign-beliefs (stms)
  (format t "~&Entered assign-beliefs~%")

  (setq *pessimistic-cost* 0)
  (loop for stm in stms do (setf (stm-unused stm) (make-list 4)))

  (setq stms (fix-for-costs stms))
  (loop with stm-list = stms
        for nxt = (a-next-level-stm stm-list) ;for each stm with pars done
        while nxt
        do (setq stm-list (remove nxt stm-list))
        do (set-aprx-costs nxt))
```

```
  ; -------Push Down Costs--------
  (if  *share* (push-down-costs stms))

  (beliefs-from-aprx-costs stms))


(defun add-rel-pform (pf)
  (loop for par in (pform-parents pf)
        do (push pf (stm-rel-child-pforms par)))
  (push pf (stm-rel-pforms (pform-child pf))))

(defun del-rel-pform (pf)
  (loop for par in (pform-parents pf)
        do (setf (stm-rel-child-pforms par)
                 (delete pf (stm-rel-child-pforms par) )))
  (setf (stm-rel-pforms (pform-child pf))
        (delete pf (stm-rel-pforms (pform-child pf)) )))

;#################################################################


;; The stms in the net are given stm-rel-pforms and stm-rel-child-pforms
;; which are the rules relevant to the current calculations.

(defun fix-for-costs (stms)
  (loop for stm in stms
        do (loop for pf in (rel-pforms stm stms)
                 do (add-rel-pform pf)))

  ;; Handle Diamond Centers
  (setq stms (create-diamond-hacks stms))

  ;; Added to deal with same-type
  (setq stms (create-same-types  stms)))


(defun create-same-types (stms)
  (loop for stm in stms
    do (loop for pf in (stm-rel-pforms stm)
         do  (let (b1 b2 st pro)
               (when (setq st
                           (loop for par in (pform-parents pf)
                                 thereis  (and (eq (car (stm-pat par)) 'same-type)
                                               par)))

                     (setq pro (same-type-prob pf))
                     ;; same-type node st has two parents: b1 and b2
                     (setq b1 (car (pform-parents (car (stm-rel-pforms st)))))
                     (setq b2 (cadr (pform-parents  (car (stm-rel-pforms st)))))

                     (cond
                      ; The case where st is :necessary
                      ((and (null b1) (null b2))
                       (setf (pform-prob pf)
                             (process-prob `((t t ,pro)) (list (pform-child pf)))))
                      (t
                       (loop for pfo in (stm-rel-child-pforms st)
                             do (del-rel-pform pfo))
                       (loop for pfo in (stm-rel-pforms st)
                             do (del-rel-pform pfo))
                       (add-rel-pform
                        (make-pform :parents (list b1 b2)
                                    :child stm
                                    :unused (make-list 3)
```

```
                                    :prob (process-prob '((t t t ,pro))
                                                        (list b1 b2))))
                          (setq stms (remove st stms))))))))
  stms)



(defun create-diamond-hacks (stms)
  (loop for stm in stms
    do (let (already-dh)

    ; If stm has more than one diamond center parent, create
    ;  only one diamond-hack.

        (loop for pf in (stm-rel-pforms stm)
          do  (let (dc new=stm npf1 npf2)
                (when (setq dc (loop for par in (pform-parents pf)
                                     thereis (and (eq (car (stm-pat par))
                                                      'diamond-center)
                                                  par)))

                ; If diamond hack already exists for stm...
                 (cond (already-dh
                        (setq npf1 (car (stm-rel-child-pforms stm)))
                        (setq new=stm (pform-child npf1))

                        (setf (pform-outs (car (stm-rel-child-pforms stm)))
                              (append
                               (pform-outs (car (stm-rel-child-pforms stm)))
                               (list (gate-equality dc))))
                        (setq npf2 (make-pform
                                    :parents (list stm (non-gate-equality dc))
                                    :child new=stm
                                    :unused (make-list 3)
                                    :prob (process-prob
                                           '((t t t 1))
                                           (list stm  (non-gate-equality dc)))))
                        (add-rel-pform npf2))

                       (t
                        (setq already-dh t)
                        (setq new=stm (make-stm :pat (list 'diamond-hack* stm)
                                                :unused (make-list 4)))
                        (push new=stm stms)
                        (setf (stm-evidence new=stm) (stm-evidence stm))
                        (setf (stm-evidence stm) nil)

                        ;; Substitute new=stm into children
                        (loop for pf in (stm-rel-child-pforms stm)
                              do (setf (pform-parents pf)
                                       (substitute new=stm stm (pform-parents pf))))

                        (setf (stm-rel-child-pforms new=stm)
                              (stm-rel-child-pforms stm))
                        (setf (stm-rel-child-pforms stm) nil)

                        ;;remove-oldstm-from-pform
                        (setq npf1 (make-pform :parents (list stm)
                                               :child new=stm
                                               :unused (make-list 3)
                                               :prob (process-prob
                                                      '((t t 1))
                                                      (list stm))))
```

```
                        (setf (pform-outs npf1) (list (gate-equality dc)))
                        (setq npf2 (make-pform
                                    :parents  (list stm (non-gate-equality dc))
                                    :child new=stm
                                    :unused (make-list 3)
                                    :prob (process-prob
                                           '((t t t 1))
                                           (list stm (non-gate-equality dc)))))
                        (add-rel-pform npf1)
                        (add-rel-pform npf2))))))))
  stms)

(defun same-type-prob (pf)
  (loop for ent in (pform-prob pf)
        thereis (and (eq (car ent) :true)
                     (eq (cadadr ent) :necessary)
                     (caddr ent))))

(defun gate-equality (dc) (car (pform-parents (car (stm-pforms dc)))))
(defun non-gate-equality (dc) (cadr (pform-parents (car (stm-pforms dc)))))

(defun rel-pforms (stm net)
  (cons (make-pform :child stm :unused (make-list 3))
                (loop for pf in (stm-pforms stm)
                      when (loop for par in (pform-parents pf)
                                 always (member par net))
                        collecting (copy-pform pf))))



(defun copy-pform (pf)  (make-pform :parents (pform-parents pf)
                                    :child (pform-child pf)
                                    :prob (pform-prob pf)
                                    :unused (make-list 3)))


(defun a-next-level-stm (stms)
  (loop for stm in stms
        thereis (and (pars-done stm) stm)))


(defun pars-done (stm)
  (every #'(lambda (pf)
             (every #'(lambda (par)
                              (loop for p in (stm-rel-pforms par)
                                    always (pform-cost p)))
                    (pform-parents pf)))
   (stm-rel-pforms stm)))

(defun set-aprx-costs (stm)
  (loop for pf in (stm-rel-pforms stm)
        do (setf (pform-cost pf) (calculate-pf-cost pf)))



  ;;Any pform who's cost is >= just assuming stm can be pruned.
  ;;(This is relevant if stm is evidence)

  (let* ((assum (loop for pf in (stm-rel-pforms stm)
                      thereis (and (null (pform-parents pf)) pf)))
         (a-cost (if assum (pform-cost assum))))
    (when assum
      (loop for pf in (stm-rel-pforms stm)
```

```
            when (and (not (eq pf assum))
                      (>= (pform-cost pf) a-cost))
              do (del-rel-pform pf))))
  (setf (stm-pass-down-figure stm) 0))


(defun calculate-pf-cost (pf)
  (cond ((pform-parents pf) (prim-pform-cost pf))
        ((and (eq (stm-evidence (pform-child pf)) :true)
              (not (real-evidence (pform-child pf)))) 0)
        (t (stm-cost (pform-child pf)))))


;*****************************************************************
                    ;PUSH-DOWN COSTS
;*****************************************************************


;; Push the costs down along the pforms and stms,  starting at assumptions,
;; down to evidence.

(defun  push-down-costs (stms)
  (let ((br  (breadth-first-traversal stms)))
    (loop for n in br  ; n will be a stm or a pform
          do (cond
               ((stm-p n)
                (stm-push-down n))
               ((pform-p n)
                (pform-push-down n))))))



;; Get a list of the stms and pforms in the net, in a breadth-first
;; ordering going from assumptions down to evidence

(defun breadth-first-traversal (stms)
  (let ((breadth () )
        (temp ()))

    ;; Start at the nodes with no children: level 1 stms.
    (loop for stm in stms
          do (cond ((null (stm-rel-child-pforms stm))
                    (setf (stm-level stm) 1)
                    (push stm temp))))

    ; And-nodes are pforms, Or-nodes are stms
    (loop for node = (pop temp)
          while node
          ; Put children at end of queue if not already there
          do (let ((in-breadth (member node breadth )) ; Already processed
                   (in-temp (member node temp ))) ; Already awaiting processing
               (cond
                 ((stm-p node)
                  (cond
                    (in-breadth
                     (setq breadth (cons node (remove node breadth)))
                     (setq temp (append temp  (stm-rel-pforms node))))
                     ((not in-temp)
                      (push node breadth)
                      (setq temp (append temp   (stm-rel-pforms node)))))
                  (cond
                   (*share*
                    (setf (stm-pass-down-figure node) (stm-cost node))
```

```
                    ; The parent stms should have their stm-level set
                    (set-stm-level (stm-rel-pforms node)
                                   (+ (stm-level node) 1)))))

                  ((pform-p node)
                   (cond
                     (in-breadth
                      (setq breadth (cons node (remove node breadth )))
                      (setq temp (append temp  (pform-parents node))))
                     ((not in-temp)
                      (push node breadth)
                      (setq temp (append temp (pform-parents node)))))

                   (if *share*
                       (setf (pform-pass-down-figure node)
                             (pform-cost node)))))))
    breadth))


;; Every next level stm gets its parents number + 1, even if it already
;; had a number.

(defun set-stm-level (pforms lev)
  (loop for pf in pforms
        do (loop for stm in (pform-parents pf)
                 do (setf (stm-level stm) lev))))


;; Push the cost of a stm down by dividing cost among the children pforms

(defun  stm-push-down (stm)
  (let ((kids  (length (stm-rel-child-pforms stm))))

    ; If stm is an assumption, then add its cost to pessimistice estimate
    (if (null (cdr (stm-rel-pforms stm)))
        (incf *pessimistic-cost* (stm-pass-down-figure stm)))
    (cond

      ((> kids 0)
       (loop for child in (stm-rel-child-pforms stm)
             do  (setf (pform-pass-down-figure child)
                       (+ (pform-pass-down-figure child)
                          (/ (stm-pass-down-figure stm) kids)))))))))



;; Push down the cost of a pform to its  child stm
(defun pform-push-down (pf)
  (let ((child (pform-child pf))
        (pdf (pform-pass-down-figure pf)))

    ; Add the cost of each pform in the net to the pessimistic cost
    (if (pform-parents pf) (incf *pessimistic-cost* (pform-cost pf)))

    (if (or (< pdf (stm-pass-down-figure child))
            (= (stm-pass-down-figure child) 0)) ;; check this out LATER***
        (setf (stm-pass-down-figure child) pdf))))



;*************************************************************
                        ; Expansions
;*************************************************************
```

```
;; Discard expansions that come from extend-partial that have
;; cost greater than  estimate + all the costs of the possible pforms.


(defun beliefs-from-aprx-costs (stms)
  (let* ((evs  (evidence-used stms))
         (agenda (insert-tree (make-partial
                               :open (make-open-pairs evs)
                               :cost (if *share*
                                         (loop for e in evs
                                               summing (stm-pass-down-figure e))
                                         0)) nil)))

    ;; The Agenda  is kept in binary tree, sorted by Partial-cost

    (loop for partial =  (poptree-least agenda)
          while partial
          with completed = nil and expansions = 0
          do (incf expansions)
          do (interum-report expansions)
          do (cond
               ((completed-p partial)
                (setq completed (merge 'list (list partial) completed
                                       #'< :key #'partial-cost)))
               (t
                (loop for e in (extend-partial partial)
                      do (setq agenda (insert-tree e agenda)))))

          until (have-best completed agenda)
          finally (let ((bests (loop for c in completed
                                     when (< (partial-cost c)
                                             (+ (partial-cost (car completed))
                                                *cost-fudge*))
                                       collecting c)))

                    (loop for s in stms
                          when (eq (car (stm-pat s)) 'diamond-hack*)
                            do (setf (stm-evidence (cadr (stm-pat s)))
                                     (stm-evidence s)))
                    (print expansions)))))
                    ;(print-info completed agenda bests expansions)))))


(defun evidence-used (stms)
  (loop for stm in stms
        when (real-evidence stm) collecting stm))

(defun interum-report (n )
  (when (= (denominator (/ n 100)) 1) (format t ".")))

(defun real-evidence (stm)
  (and (eq (stm-evidence stm) :true)          ; stm-evidence true
       (null (stm-rel-child-pforms stm))      ; no child pforms
       (loop for par in (stm-rel-pforms stm)
             thereis (pform-parents par))))


(defun have-best (completed agenda)
  (cond ((null completed) nil) ;; nothing completed
        ((or (null agenda) (null (car agenda))) t)      ;; nothing on agenda
        (t (< (+ (partial-cost (car completed)) *cost-fudge*)
              (partial-cost (tree-least agenda))))))

(defun completed-p (p)
```

```
  (when (null (partial-open p))
    (if (partial-pforms p)
        (if (not *share*)
            (setf (partial-cost p)
                  (loop for pf in (partial-pforms p)
                        summing (pform-pushed-cost pf)))
            (partial-cost p))
        (partial-cost p))))


(defun pform-pushed-cost (pf)
  (if *share* (pform-pass-down-figure pf)
      (pform-cost pf)))


(defun print-info (completed agenda bests num-exps)
  (format t "~2%Total number of expansions: ~S~%Total completed: ~S"
          num-exps (length completed))
  (format t "~%Best cost: ~S" (partial-cost (car completed)))
  (format t "~%Number of trees with about the same cost: ~S"
          (length bests))
  (multiple-value-bind
        (in-all not-in-all) (partial-disagreement bests #'partial-stms)
    (format t "~%The following stms where in every one of them:~%")
    (loop for stm in in-all do (format t "~S~%" stm))

    (when (cdr bests)
      (if not-in-all (format t "~%They disagreed on ~S~&" not-in-all)

          (multiple-value-bind
            (in-all not-in-all) (partial-disagreement bests #'partial-pforms)
            (declare (ignore in-all))
            (format t "~%Agreed on stms, disagreed on pforms: ~S~&"
                    not-in-all))))
    (check-ptrace completed agenda)))

(defun partial-disagreement (bests fn)
  (let ((in-all (loop with ans = (funcall fn (car bests))
                      for b in (cdr bests)
                      do (setq ans (intersection ans (funcall fn b) ))
                      finally (return ans))))
    (values in-all
            (loop with ans = nil
                  for b in bests
                  do (setq ans (union ans
                                      (set-difference (funcall fn b) in-all)))
                  finally (return ans)))))


;*****************************************************************
;     Extend a partial expansion from a stm
;*****************************************************************

;; Extend a partial expansion

;; Pop off the lowest level stm = the first thing in list
;; For sharing, the open list is sorted in increasing order of stm-level


(defun extend-partial (p)
  (let* ((stm (car (car (partial-open p)))))
    (cond
     ((not (member stm (partial-stms p) ))
      (cond
```

```
       (*share*
        (loop for pf in (stm-rel-pforms stm)
              when  (ok-extension p pf stm)
              collecting (make-extended-shared-partial p pf stm)))
       (t
        (loop for pf in (stm-rel-pforms stm)
              when (ok-extension p pf stm)
              collecting  (make-extended-partial p pf stm)))))

     (t (pop (partial-open p))
        (list p)))))


;; Pairs to put on the open list: stm used once
(defun make-open-pairs (stms)
  (mapcar #'(lambda (s) (cons s 1)) stms))


;; *********** Extending a partial with NO SHARING **********
;; Add new open stms to be expanded to end of partial-open list
;; Cost calculated by just adding on cost of the rule (pform traversed)

(defun make-extended-partial (p pf stm)
  (make-partial :stms  (cons stm (partial-stms p))
                :outs (append (pform-outs pf) (partial-outs p))
                :pforms (cons pf (partial-pforms p))
                :open (append (cdr (partial-open p))
                              (make-open-pairs (pform-parents pf)))
                :cost (+ (partial-cost p) (pform-cost pf))))


;; *********** Extending a partial with SHARING **********
;; Differences from non-sharing: stms that still need to be expanded
;; are merged into the open list according to their level in net, and
;; no duplicates are allowed in partial-open.
;; Cost of partial is calculated using pass-down-figures

(defun make-extended-shared-partial (p pf stm)
    (make-partial :stms  (cons stm (partial-stms p))
                :outs (append (pform-outs pf) (partial-outs p))
                :pforms (cons pf (partial-pforms p))
                :open   (insert-open (cdr (partial-open p)) (pform-parents pf))

                ;-------------------------------------------------
                ; Cost = Partial-cost + (Pass-down-figure pform)
                ;        - (Pass-down-figure stm)* children / Need-figure
                ;-------------------------------------------------
                :cost  (+ (partial-cost p)
                          (- (pform-pass-down-figure pf)
                             (* (/ (stm-pass-down-figure stm)
                                   (if (null (stm-rel-child-pforms stm))
                                       1
                                       (length (stm-rel-child-pforms stm))))
                                (cdr (car (partial-open p))))))))


;; Partial-open list is kept sorted in increasing order of stm-level
;; Each entry in open consists of a [stm  need] pair, where
;; need is the number of children stms that need that need that stm in
;; the partial so far.
```

```
(defun insert-open (open new-stms)
  (let ((new-open (copy-tree open)))
    (loop for new in new-stms
          for pos = (assoc new new-open )
          do (cond
              ; If stm is not already on open, merge according to level
              ((null pos)
               (setq new-open (merge 'list new-open
                                     (list (cons new 1)) #'<
                                     :key #'(lambda (x)(stm-level (car x))))))
              ; If already there, one more stm needs it: increase need #
              (t
               (incf (cdr pos)))))
    new-open))



(defun ok-extension (p pf stm)
  (and
    ; Don't bother expanding to something worse than worst possible cost
    (if *share* (<= (pform-pass-down-figure pf) *pessimistic-cost*)
      (<= (pform-cost pf) 1000)) ; hack so no-share doesnt run out of
                                 ; control stack for agenda

    ; Stm you want to expand is not an out
    (not (member stm (partial-outs p) ))

    ; Pform to be expanded along has no outs already in partial
    (loop for out in (pform-outs pf)
                always (not (member out (partial-stms p))))

    ; None of the parents of that pform are outs
    (loop for par in (pform-parents pf)
          always (not (member par (partial-outs p))))))


(defun prim-pform-cost (pf)
  (loop for ent in (pform-prob pf)
        thereis (and (all-true-case ent)
                     (prob-to-cost (decode-prob pf (car (last ent)))))))

(defun prob-to-cost (v) (- (log v)))

(defun all-true-case (ent)
  (and (member (pop ent) '(:true :necessary))
       (loop for rst on ent
             always (or (null (cdr rst))
                        (member (cadar ent) '(:true :necessary))))))

(defun decode-prob (pf entry)
  (let ((ans  (cond ((numberp entry) entry)
                    ((eq entry :p) (stm-cost (pform-child pf)))
                    ((functionp entry) (funcall entry pf))
                    ((and (consp entry) (functionp (car entry)))
                     (eval (cons (car entry)
                                 (loop for ent in (cdr entry)
                                       collecting
                                       (decode-prob pf ent))))))))
    (if (numberp ans) ans
        (error "Could not get pform posterior for ~S" entry))))


(defun stm-cost  (stm)
```

```
  (let ((pat (stm-pat stm))
        (pdf (stm-pass-down-figure stm)))
    (cond
      ((and *share* pdf) pdf)
     (t
       (int-more-exp stm
       (prob-to-cost
         (case (car pat)
           (inst (get (caddr pat) :inst-prior))
           (word-inst (get 'thing- :equality-prior))
           ((has-referent has-reflex-referent)
            (has-ref-hack stm))
           (same-type 1)
           (diamond-hack* 1.0e-10) ;; changed from 1e-100
           (otherwise (get 'thing- :equality-prior)))))))))


(defun int-more-exp (stm c)
  (cond ((cdr (stm-rel-pforms stm))
         (+ c 1000))
        (t c)))

(defun has-ref-hack (stm)
  (case (kind-of-justification stm)
    (pronoun .0001)
    (pp .01)
    (propernoun .02)
    (reflexive .00001)
    (else (get 'thing- :equality-prior))))


(defun kind-of-justification (stm)
  (loop for ch in (chs stm)
        thereis (let ((pat (stm-pat ch)))
                  (cond ((member (fourth pat) '(pronoun propernoun reflexive))
                         (fourth pat))
                        ((eq (second pat) 'pp) 'pp)))))

(defvar *ptrace-list* ())

(defmacro ptrace (&rest pats) `(ptrace-e ',pats))

(defun ptrace-e (pats)
  (push pats *ptrace-list*))

(defmacro unptrace (&rest pats) `(unptrace-e ',pats))

(defun unptrace-e (pats)
  (setq *ptrace-list*
        (if (null pats) nil (delete (loop for pts in *ptrace-list*
                                          thereis (and (equal pts pats) pts))
                                    *ptrace-list*))))

(defun check-ptrace (completed agenda)
  (loop for pats in *ptrace-list*
        do (let ((varbinds (retrieve (cons 'and pats))))
             (cond ((null varbinds))
                   ((cdr varbinds) (break "too many var binds"))
                   (t (check-pats-ptrace (varsub-lst pats (car varbinds))
                                         completed agenda)))))))

(defun check-pats-ptrace (pats completed agenda)
  (let ((rel-partial (or (find-rel-partial pats completed)
                         (find-rel-agenda-partial pats agenda))))
```

```
    (when rel-partial
      (format t "~&Best partial for ~S:~%~S" pats rel-partial)
      (compare-costs rel-partial
                     (cond ((not (eq rel-partial (car completed)))
                            (car completed))
                           ((cdr completed) (cadr completed))
                           (t (tree-least agenda)))))))



(defun find-rel-partial (pats partials)
  (loop for p in partials
        thereis (and (loop for pat in pats
                           always (loop for s in (partial-stms p)
                                        thereis (equal pat (stm-pat s))))
                     p)))

(defun find-rel-agenda-partial (pats agenda) nil)

(defun compare-costs (p1 p2)
  (loop for pf1 in (partial-pforms p1)
        for s1 = (pform-child pf1)
        for pf2 = (has-same-child pf1 p2)
        do (cond ((eq pf1 pf2))
                 (t (format t "~&~S: ~S ~S"
                            s1 (if *share* (pform-pass-down-figure pf1)
                                   (pform-pushed-cost pf1) )
                            (if pf2 (if *share* (pform-pass-down-figure pf2)
                                        (pform-pushed-cost pf2)) '-))))
        finally (loop for pf2 in (partial-pforms p2)
                      when (not (has-same-child pf2 p1))
                      do (format t "~&~S: -   ~S" (pform-child pf2)
                                 (if *share* (pform-pass-down-figure pf2)
                                   (pform-pushed-cost pf2))))))


(defun has-same-child (pf partial)
  (loop for pf2 in (partial-pforms partial)
        with c = (pform-child pf)
        thereis (and (eq (pform-child pf2) c) pf2)))
```

```
;; -*-Mode: Lisp;  Package: frail;  Base:10.; Syntax: Common-Lisp -*-
;; Sccsid( @(#)tree.lisp

;;*****************************************************************
;; This is the code for the binary tree that is used to store
;; the agenda of partials to be expanded.
;;*****************************************************************

(in-package 'frail)

;; Everything in the left-subtree has lower cost than root's cost
(defun left-subtree (tree)
  (cond ((null tree) nil) (t (cadr tree))))

;; Everything in the right-subtree has cost greater than or = to root's cost
(defun right-subtree (tree)
  (cond ((null tree) nil) (t (cddr tree))))


;; Insert a partial into the tree according to its cost, and return the
;; new tree
(defun insert-tree (n tree)
  (let ((nopen (if n (partial-open n) nil))
        (nstms (if n (partial-stms n) nil))
        (topen (if (car tree) (partial-open (car tree)) nil))
        (tstms (if (car tree) (partial-stms (car tree)) nil )))
    (cond
      ((null n) tree)

      ; If nothing in tree, create one
      ((or (null tree) (null (car tree)))
       (cons n (cons nil nil)))

      ; If cost of new partial is less than root, insert into left
      ; subtree.
      ((< (partial-cost n) (partial-cost (car tree)))
       (setf (cadr tree)
             (insert-tree n (left-subtree tree)))
       tree)

      ; If cost is = to cost of root, see if identical partial have
      ; been found: no need to insert new one.
      ((and (= (partial-cost n) (partial-cost (car tree)))
            (= (length nopen) (length topen))
            (= (length nstms) (length tstms))
            (subsetp nopen topen :test #'equalp)
            (subsetp nstms tstms #'equalp))
       tree)

      ; If cost of new partial is >= to root's cost, insert into
      ; right subtree.
      (t
       (setf (cddr tree)
             (insert-tree n (right-subtree tree)))
       tree))))



;; Return the partial with the least cost in the the tree
;; (without removing it). This is the left-most leaf in the tree.
(defun tree-least (tree)
  (cond
    ((or (null tree) (null (car tree)))
     nil)
```

```
    ((null (left-subtree tree))
     (car tree))
    (t
     (tree-least (left-subtree tree)))))


;; Remove and return the partial with the least cost in the tree.
(defun poptree-least (tree)
  (cond
    ((or (null tree) (null (car tree)))  nil)
    ((null (left-subtree tree))
     (prog1 (car tree)
            (setf (car tree) (car (right-subtree tree)))
            (setf (cdr tree) (cdr (right-subtree tree)))))
    ;If next level down has no left subtree, then stop now
    ((null (left-subtree (left-subtree tree)))
     (prog1 (car (left-subtree tree))
            (setf (cadr tree) (right-subtree (left-subtree tree)))))
    ;Otherwise get the least-cost item in the left-subtree
    (t (poptree-least (left-subtree tree)))))
```