An Efficient Viterbi Algorithm on DBNs

Wei Hu, Yimin Zhang, Qian Diao, Shan Huang

Intel China Research Center
Beijing, P. R. China
{wei.hu, yimin.zhang, qian.diao, shan.huang}@intel.com

10.21437/Eurospeech.2003-694

Abstract

DBNs (Dynamic Bayesian Networks) [1] are powerful tool in modeling time-series data, and have been used in speech recognition recently [2,3,4]. The "decoding" task in speech recognition means to find the viterbi path [5](in graphical model community, "viterbi path" has the same meaning as MPE "Most Probable Explanation") for a given acoustic observations. In this paper we describe a new algorithm utilizes a new data structure "backpointer", which is produced in the "marginalization" procedure in probability inference. With these backpointers, the viterbi path can be found in a simple backtracking. We first introduce the concept of backpointer and backtracking; then give the algorithm to compute the viterbi path for DBNs based on backpointer and backtracking. We prove that the new algorithm is correct, faster and more memory saving comparison with old algorithm. Several experiments are conducted to demonstrate the effectiveness of the algorithm on several well known DBNs. We also test the algorithm on a real world DBN model that can recognize continuous digit numbers.

1. Introduction

Based on junction tree inference algorithm [6,7,8], A.P. David [6,7] gave the first algorithm to get the MPE in static BNs. We call it as "DAP algorithm". It first performs a "collect evidence" procedure (forward pass) using a "MAX flow"[6,7]; then find the maximal entry in the root clique's potential and set it to 1, set other entries to 0s; In the backward pass, DAP algorithm "distribute evidence" based on the changed root potential and in the same time get the MPE. K. Murphy extended the idea to DBNs based on his "1.5 slice junction tree algorithm" [1,9]. We call it as KPM algorithm and will introduce in later section.

KPM algorithm suffers from two problems. First it need a full "backward pass" procedure, this is slow, and has computational complexity not only depend on the structure of the junction tree, but also the size (number of entry) of each cliques' potentials in junction tree. Furthermore it requires all cliques and separators potentials in the "backward pass". All the potentials produced in the "forward pass" need to be saved. This will cause great overhead not only on memory, but also performance of the algorithm. We modify KPM algorithm by introducing data structure "backpointer" and function "backtracking". See section 3 for algorithm description and experiment results in section 4.

In this paper we first introduce concepts of "backpointer" and "backtracking", see section 2. Section 3 presents the KPM algorithm, the new algorithm. In section 4 we list some experiments results on both algorithms, compare the performance and memory consummation on several well known DBNs, and on a real world two stream speech DBN models. Section 5 is the conclusion.

2. Backpointer and Backtracking

2.1. Backpointer

The idea of "backpointer" has long been used in speech recognition, where it is used to do backtracking in decoding of HMMs to get the viterbi path [5]. Here we define a "backpointer" as an integer array produced by a "max marginalization" procedure (a "max marginalization" means to pass a "max flow" when perform a marginalization), each element in a backpointer is the index of a configuration (entry) in the potential that be marginalized.

For example, potential $\boldsymbol{P_1}$ contain 3 binary variables $\boldsymbol{X_1}$, $\boldsymbol{X_2}$ and $\boldsymbol{X_3}$ (We say that $\boldsymbol{P_1}$ has "domain" $\boldsymbol{X_1}$, $\boldsymbol{X_2}$ and $\boldsymbol{X_3}$), and has joint probability distribution $\boldsymbol{p(X_1, X_2, X_3)} \leftarrow$ [.15, .20, .05, .12, .27, .03, .11, .07]. While a marginalization of $\boldsymbol{P_1}$ to produce $\boldsymbol{P_2}$ that is a potential on variables $\boldsymbol{X_1}$ and $\boldsymbol{X_3}$ ($\boldsymbol{P_2}$ has "domain" $\boldsymbol{X_1}$ and $\boldsymbol{X_3}$), i.e. we marginalize probability distribution $\boldsymbol{p(X_1, X_2, X_3)}$ to probability distribution $\boldsymbol{p(X_1, X_3)}$. We get the potential $\boldsymbol{P_2}$ with distribution is [.15, .20, .27, .07]. At the same time, we also get the corresponding backpointer which is [1, 2, 5, 8]. The first element "1" in the backpointer means that the first element in $\boldsymbol{P_2}$ –-".15" is passed from the 1st element of $\boldsymbol{P_1}$. Similarly, the third element "5" in the backpointer means that the third element of $\boldsymbol{P_2}$ is passed from the 5th element—".27" from $\boldsymbol{P_1}$. A backpointer has the same size and shape as the produced result potential in a max marginalization operation except that they have different data type, i.e. integer instead of floating type.

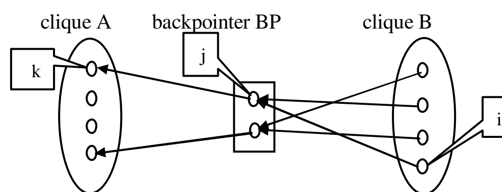

Fig.1 Backpointer and BackTracking

2.2. Backtracking

A backtracking procedure that utilizes a backpointer between a pair of cliques is illustrated in Fig.1. Where clique $\boldsymbol{A}$ marginalizes to separator $\boldsymbol{S}$ (not shown in Fig.1), produce a backpointer $\boldsymbol{BP}$; then multiply $\boldsymbol{S}$ to clique $\boldsymbol{B}$. Now we want to find the entry in $\boldsymbol{A}$ that corresponds to an entry "$\boldsymbol{i}$" in $\boldsymbol{B}$. First we find the entry in $\boldsymbol{BP}$ that corresponds to "$\boldsymbol{i}$" in $\boldsymbol{B}$. For a given entry in $\boldsymbol{B}$, there is one and just one corresponding entry in $\boldsymbol{S}$ (see the definition of potential multiplication in a junction tree), which is determined by

domain of ***B*** and ***S***. In this case, the entry in ***S*** that corresponding to entry "*i*" in ***B*** is "*j*", as illustrated in Fig.1. From ***BP*** we get $k \leftarrow BP_j$, meaning that the corresponding entry in ***A*** is "*k*" for entry "*i*". Fig.2a present the algorithm for backtracking between a pair of neighbor cliques.

BACKTRACK_IN_PAIR(***Parent, Child, BP, idx***)
1 $i \leftarrow idx$;
2 $j \leftarrow$ CLIQUE_TO_BACKPOINTER(***Parent, BP, i***);
3 $k \leftarrow BP_j$;
4 **return** ***k***

Fig.2a backtracking in a pair of cliques

BACKTRACK_IN_TREE(***TREE, BP_TREE, idx***)
1 $RESULT_{TREE.ROOT} \leftarrow idx$;
2 **for** ***Parent*** $\leftarrow$ ***TREE.PARENT***
3 **for** ***Child*** $\leftarrow$ $TREE.CHILDREN_{Parent}$
4 $BP \leftarrow BP_TREE_{[Parent,\ Child]}$;
5 $idx \leftarrow RESULT_{Parent}$;
6 $RESULT_{Child} \leftarrow$
BACKTRACK_IN_PAIR(***Parent, Child, BP, idx***)
7 **return** ***RESULT***

Fig.2b backtracking in a tree

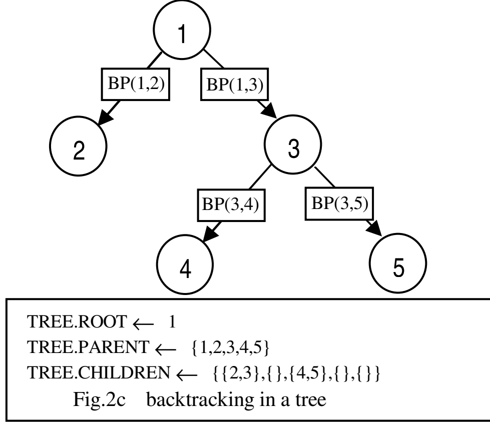

Fig.2c backtracking in a tree

In Fig.2a "***Parent***" and "***Child***" are two cliques; "***BP***" is the backpointer between the "***Parent***" and "***Child***"; "***idx***" is the index of an entry in "***Parent***" that need be backtracked. Function "CLIQUE_TO_BACKPOINTER" just computes entry "*j*" in ***BP*** that correspond to entry "*i*" in "***Parent***". As illustrated in Fig.1. Backtracking can easily be extended to backtrack through a chain of cliques, or to backtrack in a tree. Backtracking in a tree means given an entry of the root clique, perform a backtracking level by level, eventually reach each leaf cliques, and record each corresponding entries in each cliques. Fig.2b and Fig.2c illustrate to perform a backtracking in a tree. The ***BP_TREE*** is the backpointers of the tree, which is a 2D array.

All computations in the backtrackings are integer type and independent on the size of each clique's potentials. This is different with potential's multiplication, division and marginalization which is floating point computation and has complexity known as the direct ratio to size of each potentials in operation. Briefly backtracking in a pair of potentials has constant complexity **O(1)**, but the backward pass in a pair of potentials has complexity **O(|A| + |B|)**, where **|A|**, **|B|** are sizes of **A** and **B** respectively. And in the same time, backtracking need store **|S|** integer data, but backward pass need store (**|A| +|B| +|S|**) floating data.

2.3. Proof of Backtracking

The DAP algorithm has been proved in [6]. The KPM algorithm is also correct [1]. We can prove that the backtracking based on backpointer can produce the same result as the DAP algorithm.

Consider a simple junction tree that has only two cliques ***A*** and ***B***, where ***A*** is the leaf and ***B*** is the root, ***S*** is the separator, (Fig.1). In the forward pass, we get the backpointer ***BP***. $BP_j \leftarrow k$ means that the kth element in ***A*** passed its value to the jth element in ***S***. The arrow from the ith element of ***B*** to the jth element of ***S*** is not backpointer, but a deterministic arrow. There is one and only one element in ***S*** corresponding to a specific element in ***B*** (for more detail, see the description of junction tree algorithm). Now assume the ith element is the maximal element in ***B***.

From the algorithm in Fig.2a we know that the configuration of the kth element in ***A*** will become a part of the MPE. We now prove that the kth element is still the element we need to find when performing the DAP algorithm. Denote POT_X as the potential of a clique or a separator **X**, and POT_X^k as the kth entry of POT_X. Now suppose the ith element in ***B*** is the maximal, and set POT_B^i to 1, set all other entry to 0s. When performing "distribute evidence", the DAP algorithm do:

1 $POT_A \leftarrow POT_A / POT_S$
2 $POT_S \leftarrow$ MARGINALIZE(POT_B)
3 $POT_A \leftarrow POT_A * POT_S$

We know $BP_j \leftarrow k$, means that POT_A^k is the maximal element in $POT_A^{CONS(j)}$, where ***CONS(j)*** indicate a set of elements that "consistent with" ***j*** (consistent means that two elements have the same configuration in same domains) and $k \in CONS(j)$. After step 1, the POT_A^k is still the maximal element in $POT_A^{CONS(j)}$, and has the value 1, because all elements in $POT_A^{CONS(j)}$ need to divide by same POT_S^j. After step 2, POT_S^j equals to 1 (passed from the ith element in ***B***). Then after step 3, the element POT_A^k is still the maximal element in $POT_A^{CONS(j)}$ because all elements in $POT_A^{CONS(j)}$ will multiply a const value 1, and we have already known that the original POT_A^k is the maximal in $POT_A^{CONS(j)}$ after step 1. Thus we know that the DAP algorithm and the backpointer based backtracking algorithm can produce the same MPE result. This procedure can similarly be generalized to junction tree with more than two cliques or a junction trees chain produced from a DBN.

3. Compute MPE on DBNs

3.1. The KPM Algorithm

KPM algorithm produces an independent junction tree for each time slice; all these junction trees have the same structure except the first. Then glue these junction trees by the "forward interface", see Fig.3. A "forward interface" means variables in a single time slice in the DBN that has an arrow pointer to later time slice variables. In Fig.3, J_1 to J_T are the T junction trees induced from the original DBN, I_1 to I_{T-1} are the T-1 interface cliques for corresponding junction trees. Clique D_t is the input clique in J_t, C_t is the root and output clique of J_t. See Fig.4 for KPM algorithm.

KPM_ALGORITHM(***DBN, EVIDENCE***)
1 $JTREES_{1:T} \leftarrow$ CONSTRUCT_JTREES_FROM_DBN(***DBN***);
2 $\{CP_1, SP_1\} \leftarrow$

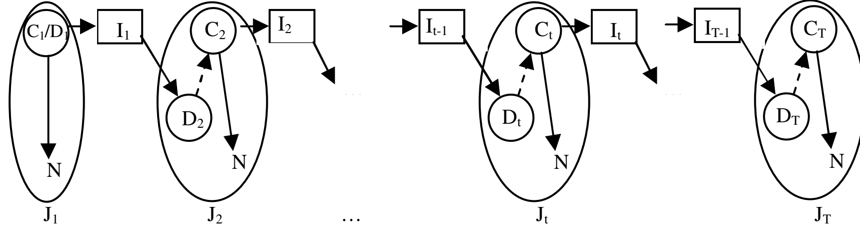

Fig.3 The glued junction trees of DBN inference algorithm

INITIALIZE_JTREE($\boldsymbol{JTREES_1}$, $\boldsymbol{DBN}$, $\boldsymbol{EVIDENCE_1}$);
3 $\{CP_1, SP_1\} \leftarrow$ COLLECT_EVIDENCE($\boldsymbol{JTREES_1}$, $\boldsymbol{CP_1}$, $\boldsymbol{SP_1}$);
4 $IP_1 \leftarrow$ MARGINALIZE(CP_1^C);
5 **for** $t \leftarrow 2 : T$
6 $\{CP_t, SP_t\} \leftarrow$
INITIALIZE_JTREE($\boldsymbol{JTREES_t}$, $\boldsymbol{DBN}$, $\boldsymbol{EVIDENCE_t}$);
7 $CP_T^D \leftarrow CP_T^D * IP_{t-1}$;
8 $\{ CP_t, SP_t \} \leftarrow$
COLLECT_EVIDENCE($\boldsymbol{JTREES_t}$, $\boldsymbol{CP_t}$, $\boldsymbol{SP_t}$);
9 $IP_t \leftarrow$ MARGINALIZE(CP_T^C);
10 $\{CP_T^C, MPE_T^C\} \leftarrow$ SET_MAX_CONFIG(CP_T^C);
11 **for** $t \leftarrow T$: 1
12 $CP_t \leftarrow$ DISTRIBUTE_EVIDENCE($\boldsymbol{JTREES_t}$, $\boldsymbol{CP_t}$, $\boldsymbol{SP_t}$);
13 $MPE_t \leftarrow$ FIND_MAX_CONFIG(CP_t);
14 $CP_{t-1}^C \leftarrow CP_{t-1}^C / IP_{t-1}$;
15 $IP_{t-1} \leftarrow$ MARGINALIZE(CP_t^D);
16 $CP_{t-1}^C \leftarrow CP_{t-1}^C * IP_{t-1}$;
17 $VITERBIPATH_t \leftarrow$
CONFIG_TO_INDEX($\boldsymbol{JTREES_t}$, $\boldsymbol{MPE_t}$)
18 **return** ***VITERBIPATH***

Fig.4 KPM algorithm to find MPE in DBN

Where $\boldsymbol{CP_1}$, $\boldsymbol{SP_1}$ are the cliques and separators' potentials in time t; $\boldsymbol{IP_t}$ is the interface potential in time t; $\boldsymbol{CP_t^C}$ is the potential of clique $\boldsymbol{C}$ in $\boldsymbol{JTREE_t}$. KPM algorithm is a natural extension of the DAP algorithm from BNs to DBNs. First it performs a forward pass (line 2—9) from time 1 to time T, computes and stores all the cliques, separators and interfaces' potentials. It then finds and sets the max configuration of the root clique of time T. At last, it performs a backward pass from time T to 1 (line 11--17), based on the potentials that computed in the forward pass, and gets the MPE of all the variables in each time slice.

As illustrated in Fig.4, KPM algorithm suffered from the same problems as the DAP algorithm. First it needs to store all the potentials from time 1 to time T in the forward pass, this needs a lot of memory. Second, operations on potential level are complex floating point computation with complexity known as the size of the potentials, which will make the backward pass slow. Both problems can be solved in the below new algorithm.

3.2. The New Algorithm

With introducing "backpointers" into the KPM algorithm, and performing a backtracking instead of a backward pass, we get a new algorithm that is more efficient. See Fig.5.

NEW_ALGORITHM($\boldsymbol{DBN}$, $\boldsymbol{EVIDENCE}$)
1 $JTREES_{1:T} \leftarrow$ CONSTRUCT_JTREES_FROM_DBN($\boldsymbol{DBN}$);
2 $\{CP_1, SP_1\} \leftarrow$
INITIALIZE_JTREE($\boldsymbol{JTREES_1}$, $\boldsymbol{DBN}$, $\boldsymbol{EVIDENCE_1}$);
3 $\{CP_1, SP_1, BPS_1\} \leftarrow$
COLLECT_EVIDENCE_BP($\boldsymbol{JTREES_1}$, $\boldsymbol{CP_1}$, $\boldsymbol{SP_1}$);
4 $\{IP_1, BPI_1\} \leftarrow$ MARGINALIZE_BP(CP_1^C);
5 **delete** CP_1, SP_1;
6 **for** $t \leftarrow 2 : T$
7 $\{CP_t, SP_t\} \leftarrow$
INITIALIZE_JTREE($\boldsymbol{JTREES_t}$, $\boldsymbol{DBN}$, $\boldsymbol{EVIDENCE_t}$);
8 $CP_T^D \leftarrow CP_T^D * IP_{t-1}$;
9 **delete** IP_{t-1};
9 $\{CP_t, SP_t, BPS_t\} \leftarrow$
COLLECT_EVIDENCE_BP($\boldsymbol{JTREES_t}$, $\boldsymbol{CP_t}$, $\boldsymbol{SP_t}$);
10 $\{IP_t, BPI_t\} \leftarrow$ MARGINALIZE_BP(CP_T^C);
11 **delete** CP_t, SP_t;
12 $\{MPE_T^C, idx\} \leftarrow$ FIND_MAX_CONFIG(CP_T^C);
13 **for** $t \leftarrow T$: 1
14 $OFFSET_t \leftarrow$
BACKTRACK_IN_TREE($\boldsymbol{JTREES_t}$, $\boldsymbol{BPS_t}$, $\boldsymbol{idx}$);
15 $MPE_t \leftarrow$ INDEX_TO_CONFIG($\boldsymbol{OFFSET_t}$);
16 $idx \leftarrow$ BACKTRACK_IN_PAIR($\boldsymbol{D_t}$, $\boldsymbol{C_{t-1}}$, $\boldsymbol{BPI_{t-1}}$, $\boldsymbol{OFFSET_t}$);
17 $VITERBIPATH_t \leftarrow$
CONFIG_TO_INDEX($\boldsymbol{JTREES_t}$, $\boldsymbol{MPE_t}$)
18 **return** ***VITERBIPATH***

Fig.5 New algorithm to find MPE in DBN

Where $\boldsymbol{BPS_t}$, $\boldsymbol{BPI_t}$ are the backpointers for $\boldsymbol{JTREES_t}$ and interface clique $\boldsymbol{I_t}$ respectively. The function "COLLECT_ EVIDENCE_BP" is the same as the function "COLLECT_EVIDENCE" in Fig.4, except it produces the additional "backpointer" in each marginalization.

The function "BACKTRACK_IN_TREE" perform a backtracking in a single junction tree which is described in Fig.2b. Function "BACKTRACK_IN_PAIR" is just a simple backtracking from clique $\boldsymbol{D_t}$ to clique $\boldsymbol{C_{t-1}}$ through the interface $\boldsymbol{I_{t-1}}$, using the backpointer $\boldsymbol{BPI_t}$.

We can see that the backtracking (line 13 – 17) need not use any potentials values. All the computations are the simple integer operations and have constant complexity. Memory used to store the backpointers will be much less than to store all the potentials. We generalize the Viterbi backtracking algorithm to junction trees. The basic idea is that instead of keeping a single integer to represent the backpointer, one keeps a vector of integers, which encode the "argmax" computed in the forwards pass.

For example, consider passing a message from potential $\boldsymbol{P_1(X_1, X_2, X_3)}$ to $\boldsymbol{P_2(X_1, X_3)}$ in the forwards pass (i.e. marginalization of $\boldsymbol{P_1}$ to $\boldsymbol{P_2}$). The message, which will be stored on the separator, can be computed as

$$MESSAGE(X_1, X_3) \leftarrow \mathrm{MAX}_{X2}(P_1(X_1, X_2, X_3))$$

We also store the backpointer,

$$BP(X_1, X_3) \leftarrow \mathrm{ARGMAX}_{X2}(P_1(X_1, X_2, X_3))$$

On the backward pass, suppose (by induction) that (X_1^*, X_3^*) is the optimal assignment to the separator. Then we can compute the optimal assignment to $\boldsymbol{P_1}$ by setting

$X_2^* \leftarrow BP(X_1^*, X_3^*)$

This algorithm requires keeping backpointers for every separator and interface in each time slice. However, old clique potentials do not need to be stored, just as old alpha variables are not needed in Viterbi for HMMs [5]

4. Experiments Results

Experiments are conducted on a PC with a Pentium III XEON 933 MHz and 512M memory. Algorithms mentioned above are implemented based on BNT[9].

4.1. Experiments on 7 Typical DBNs

Experiments are conducted on 7 typical sample DBN models (all are available in BNT, "chmm(m,n)" means that the coupled HMM has "m" hidden chains and each hidden variable has "n" states). All have 10 time slices. We compare the total execution time and the maximal memory used when running both algorithms on each DBNs. See Table 1.

	Execute time(s)		Memory used(KB)	
	new	KPM	new	KPM
mildew	0.234	0.297	171	394
water	0.297	0.422	247	905
chmm(9,2)	0.657	0.859	423	2,138
chmm(4,5)	1.001	1.500	270	7,214
BAT	1.031	1.109	1,282	4,429
chmm(10,2)	1.016	1.172	516	4,274
chmm(5,5)	5.125	8.344	552	41,953

Table 1, KPM algorithm and the new algorithm on 7 typical DBNs

4.2. A Real DBN Application on Speech Recognition

DBNs have been used in speech recognition in recent years [1,2,3]. In this experiment, we construct a DBN to recognize continuous digit strings (AURORA data). Other than normal implementations, we use 2-asynchronous observation streams to model the asynchronous property of extracted features. For simplicity, we use single Gaussian to represent the acoustic observations, and use a whole word model to represent each digit number '0'-'9'and the 'silence'. Each digit number has 16 states, and 'silence' has three states.

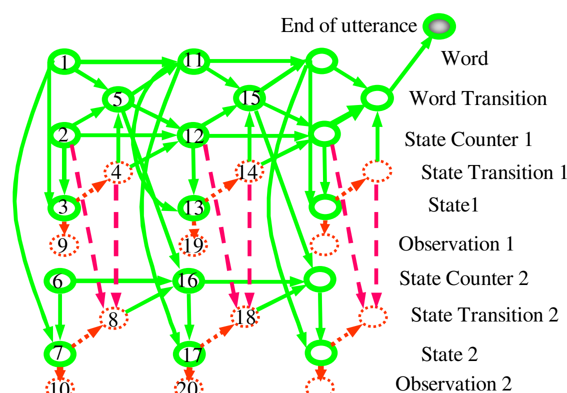

Fig.6, the decoding network

In Fig.6 nodes with dashed circle are trainable; nodes with green circle are deterministic and can not be train. This is a two stream model, with master stream (state 1) determine when to do word transition, and the slave stream (state 2) can not differ with master stream more than 1 state. The node orders for first and second slice are shown in the figure. Nodes 1-8 are discrete nodes, and nodes 9 and 10 are 39-dimensional single Gaussian observation nodes. Detail can refer to [4].

Network in Fig.6 will produce junction trees with very big cliques; for example one of the cliques has more than 1.83 million entries. If we use KPM algorithm to compute the MPE, the overhead of store all the potentials will be too large to afford, prevent it to be used in longer time slice. Table 2 shows the execute time and memory consummation for KPM algorithm and algorithm 2 for 5, 10 and 20 time slices respectively.

	Execute time(s)		Memory used(KB)	
	new	KPM	new	KPM
T = 5	8.89	17.39	2,017	97,104
T = 10	18.69	37.58	4,203	199,194
T = 20	37.98	141.17	8,534	403,375

Table 2, KPM algorithm and new algorithm on speech recognition

From Table 1 we see that our new algorithm can get 10% --50% faster than KPM algorithm, and can save memory about 80%--90% in the same time. Table 2 show us that the new algorithm can get higher speedup factor (>2) compare to table 1, because the model is so large, need so many memory to store all the potentials, and so many memory access makes the KPM algorithm's performance degrade greatly than it should be. The situation of KPM algorithm becomes worse with the increasing of time slice of a DBN.

5. Conclusion

We proposed a new algorithm that can compute the viterbi path for DBNs, based on a new introduced data structure we called a "backpointer". We prove that the new algorithm is correct. Experiments show that the new algorithm will get 10%--40% faster and save 80%--90% memory than old algorithm. The new algorithm can get even faster (>2X) than old algorithm when memory issue become much critical.

Reference

[1] Kevin Murphy, *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, U.C. Berkeley, Dept. Comp. Sci., 2002.

[2] G. Zweig, "*DBN based speech recognition*", Ph.D. thesis, U.C. Berkeley, 1998.

[3] J. Bilmes, G. Zweig, *"The graphical models toolkit: an open source software system for speech and time-series processing*", ICASSP Proc, 2002.

[4] Y. Zhang, et al., "*DBN based multi-stream model for speech"*, ICASSP Proc, 2003.

[5] L. R. Rabiner, *"A tutorial on Hidden Markov Models and selected applications in Speech Recognition"*, Proc. of the IEEE, 77(2):257-286, 1989.

[6] A. P. Dawid, *"Applications of a general propagation algorithm for probabilistic expert systems",* Statistics and Computing, 2, 25-36, 1992.

[7] R.G.Cowell, et.al., *Probabilistic Networks and Expert Systems,* Springer, pp. 97-98, 1999.

[8] C. Huang and A. Darwiche, *"Inference in Belief Networks: A procedural guide"*, Intl. J. Approximate Reasoning, 15(3):225-263, 1996.

[9] Kevin Murphy, *"The Bayes Net toolbox for Matlab*", Computing Science and Statistics, vol 33, 2001.