The Juicer LVCSR Decoder - User Manual for Juicer version 0.5.0

Darren Moore [a]

IDIAP–Com 06-03

October 26, 2005

[a] IDIAP Research Institute and Ecole Polytechnique Federale de Lausanne (EPFL), Martigny, Switzerland

Chapter 1

Introduction

1.1 Overview

Juicer is a decoder for HMM-based large vocabulary speech recognition that uses a weighted finite state transducer (WFST) representation of the search space. The package consists of a number of command line utilities: the Juicer decoder itself, along with a number of tools and scripts that are used to combine the various ASR knowledge sources (language model, pronunciation dictionary, acoustic models) into a single, optimised WFST that is input to the decoder. The Juicer package is distributed under a BSD license (see Appendix C).

The major advantages of the WFST-based approach is the decoupling of the decoding network from the decoding engine, as well as a common representation of the various ASR knowledge sources allowing standardised techniques to be used for constructing a complete, optimised search space. These characteristics allow straightforward incorporation of new capabilities, such as decoding with a custom grammar or non-standard lexical constraints, without requiring modification of the decoding engine.

An overview of Juicer is illustrated in Figure 1.1.

This document contains detailed instructions for use. Those wishing to attempt a "quick start" can skip to Chapter 4 and examine the example systems and decoding configurations provided.

1.2 Current capabilities

- Language Modelling
  - simple word-loop with start/end silence
  - word-pair grammar
  - N-gram (arbitrary N, subject to memory limitations - see Chapter 5)
- Acoustic Modelling
  - monophones
  - word-internal n-phones (tri/quin/...)
  - cross-word triphones
  - HTK MMF file format support
- Dictionary
  - multiple pronunciations of same word

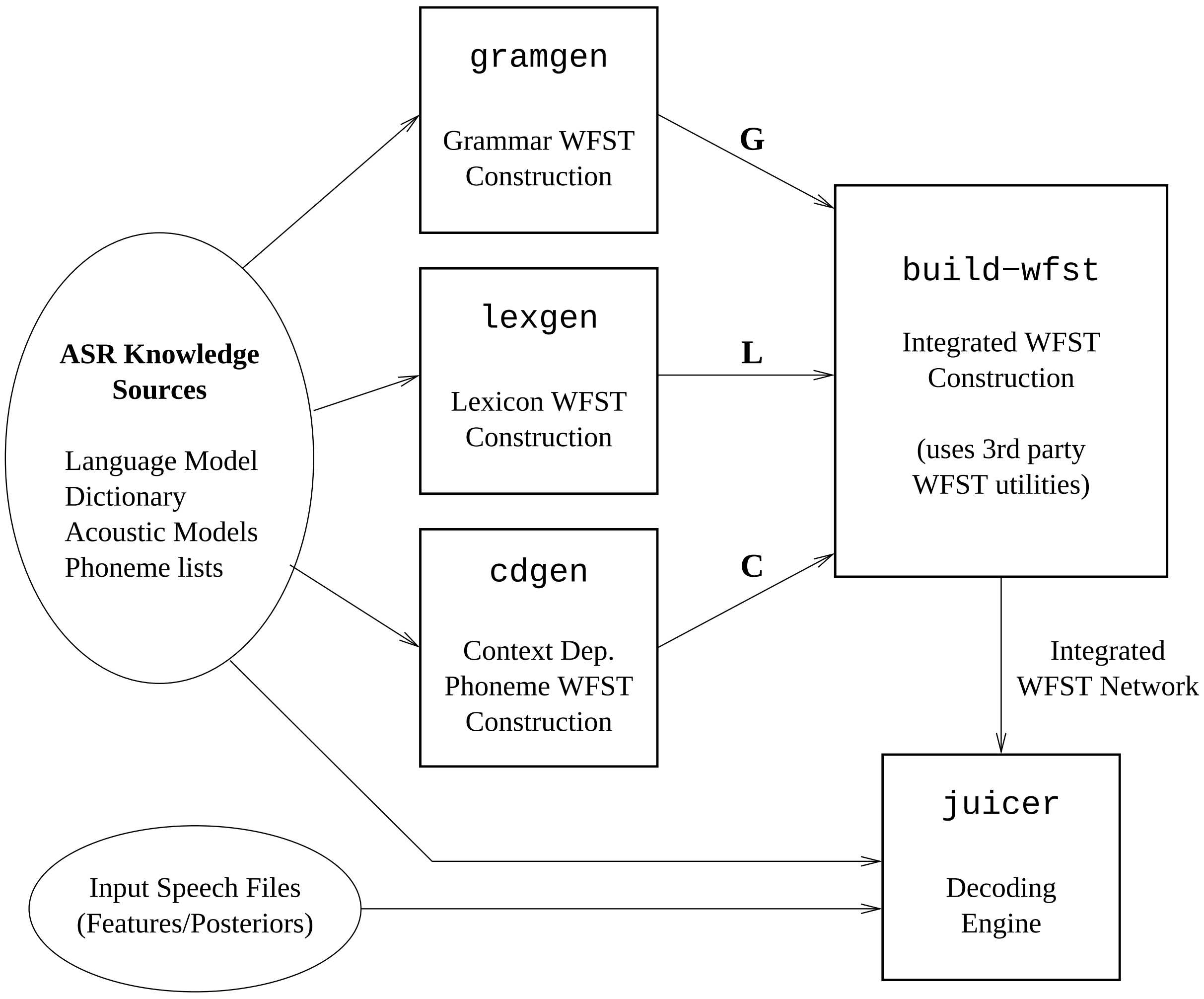
gramgen
Grammar WFST Construction
lexgen
Lexicon WFST Construction
cdgen
Context Dep. Phoneme WFST Construction
G
L
C
build-wfst
Integrated WFST Construction
(uses 3rd party WFST utilities)
ASR Knowledge Sources
Language Model
Dictionary
Acoustic Models
Phoneme lists
Integrated WFST Network
juicer
Decoding Engine
Input Speech Files (Features/Posteriors)

Figure 1.1: High level architecture of the Juicer decoding package

  - pronunciation probabilities

- Decoding Search
  - Flexible WFST-based decoder (decoding network independent of decoding engine implementation)
  - Viterbi search
  - Beam-search pruning (global, model-end)
  - Histogram pruning
  - Lattice generation
  - Word-level or model-level output

- Other
  - Hybrid HMM/ANN decoding supported (using LNA-format posterior files).

Chapter 2

Installation

- Copy.
    1. Create a directory, `<dir>`, for juicer (e.g. `/home/moore/juicer`)
    2. `cp /home/speech/moore/juicer_rel/juicer-0.5.0.tar.gz <dir>`
- Extract files.
    1. `cd <dir>`
    2. `tar -xzvf juicer-0.5.0.tar.gz`
- Build Juicer tools.
    1. `cd <dir>/code/tools`
    2. `make`
    3. Try running tools as a test for build success (e.g. `./cdgen -h`)
- Build MIT FST toolkit.
    1. `cd <dir>/tools/mitfst-1.0.0/src/lib`
    2. `make`
    3. `cd ../tools`
    4. `make`
    5. Try running tools as a test for build success (e.g. `../../bin/fst_info`)
- Obtain AT&T FSM toolkit version 4.0.
    1. Download from `http://www.research.att.com/sw/tools/fsm/`
    2. Install in a directory, `<fsmdir>`, of your choice.
- Configure juicer environment file.
    1. edit `<dir>/code/tools/juicer-env.csh`
    2. modify `JUTOOLS` path to be absolute path of `<dir>/code/tools`
    3. modify `ATTFSMDIR` path to be absolute path of `<fsmdir>/bin`

Chapter 3

Instructions for use

Decoding with Juicer involves 3 main steps :

1. Generate 3 individual WFST's (LM, lexicon, phoneme).
2. Build integrated WFST from individual WFST's.
3. Run Juicer decoder.

3.1 WFST generation

This section describes the three tools: `gramgen`, `lexgen`, and `cdgen` that are used to transform the ASR knowledge sources into weighted finite-state transducer representations. The output of each of these tools is 3 files: an FSM format file [1] containing the WFST itself, along with input and output symbols files that map from the numerical labels in the FSM file to the relevant string representations (words, phoneme names, etc).

It is important to maintain consistency in the files and options that are passed to these tools, and subsequently to the juicer decoder. For example, the `lexgen` and `cdgen` tools as well as the `juicer` decoder all have a `-monoListFName` option. The same monophone list file must be passed to all these tools for correct operation.

One rudimentary check that consistency has been preserved between the `gramgen` and `lexgen` tools is to compare the output symbols file generated with `gramgen` to the input symbols file generated with `lexgen`. These two symbols files should be identical. The same comparison can be done with the output symbols file from `lexgen` and the input symbols file from `cdgen`.

3.1.1 Grammar WFST

The grammar WFST is generated using the `gramgen` executable, which can be found in the tools directory after the source code has been successfully built. The purpose of the grammar WFST is to ensure that sequences of recognised words conform to the grammar or language model.

The command-line help (obtained by executing `gramgen -h`) is displayed in Figure 3.1. Basically, the user inputs a dictionary (defining the set of words to recognise), the type of grammar to use, and depending on the grammar type, a file containing the grammar definition (e.g. an ARPA LM file when the grammar type is `ngram`). The tool then outputs a WFST in AT&T FSM format, together with input and output symbol files.

The tool is capable of producing grammar WFST's of the following types:

- Simple word loop, with sentence marker words at beginning and end.
- Word-pair grammar.

```
#
# usage: gramgen [options]
#

Lexicon Options:
  -lexFName <string>      -> the dictionary filename []
  -sentStartWord <string> -> the name of the dictionary word that will start every sentence []
  -sentEndWord <string>   -> the name of the dictionary word that will end every sentence []

Language Model Options:
  -lmFName <string>       -> the LM file (in ARPA LM format or BBN word-pair format) []
  -lmScaleFactor <real>   -> the factor by which log LM probs are scaled during decoding [1]
  -wordInsPen <real>      -> word insertion penalty [0]
  -unkWord <string>       -> the name of the unknown word in the LM []

Transducer Options:
  -gramType <string>      -> the type of grammar to generate (silwordloopsil,ngram,wordpair) []
  -fsmFName <string>      -> the FSM output filename []
  -inSymsFName <string>   -> the input symbols output filename []
  -outSymsFName <string>  -> the output symbols output filename []
  -genTestSeqs            -> generates some test sequences from the grammar transducer just constructed.
```

Figure 3.1: Command line help message for the `gramgen` tool.

- N-gram language model.

The desired word insertion penalty and language model scaling factor are specified during grammar WFST construction. These parameters cannot be specified when invoking the decoder.

The `gramgen` command line options are described in detail in Table 3.1.

Table 3.1: Detailed description of `gramgen` command line options.

Option	Description
`-lexFName` *string*	**Required.** The name of the text file containing the pronunciation dictionary.
`-sentStartWord` *string*	The word in the dictionary that begins every recognised sentence (e.g. `<s>`). Note: this option is mandatory for grammar types `silwordloopsil` and `wordpair`.
`-sentEndWord` *string*	The word in the dictionary that ends every recognised sentence (e.g. `</s>`). Note: this option is mandatory for grammar types `silwordloopsil` and `wordpair`.
`-lmFName` *string*	The file containing the original grammar specification for grammar types of `ngram` and `wordpair`. N-gram LM files are expected to be in standard ARPA-MIT format (see Section 16.7.1 of [7]). Word pair grammar files are expected to be in the BBN word pair grammar format as used in the ARPA Navel Resource Management task. Note: the option is mandatory for grammar types `ngram` and `wordpair`, and is ignored otherwise.
`-lmScaleFactor` *float*	All LM probabilities from the original grammar specification are converted to natural log, and are then multiplied by this value. The scaled values are used in the output WFST. The purpose of this parameter is to bring LM weights into line with acoustic model output likelihoods (which are typically grossly under-estimated), and also to control the importance of the acoustic models relative to the language model. This option is ignored for grammar type `silwordloopsil`. Default value if omitted is 1.0.

Table 3.1: (continued)

Option	Description
`-wordInsPen` *float*	This is a natural log value that is added to each LM probability (after scaling). The general effect of varying this parameter is to tradeoff between the insertion and deletion rates in the final decoder output. Note that the LM scaling factor also affects insertion and deletion rates. Default value if omitted is 0.0.
`-unkWord` *string*	This only applies when grammar type is `ngram` and specifies the unknown word in the ARPA LM file (e.g. `<unk>`). All dictionary words that are not present in the LM are mapped to the unknown word for the purpose of determining LM probabilities. Note that an error occurs if there are words in the LM that are not in the dictionary.
`-gramType` *string*	**Required.** This specifies the type of the grammar that is to be output. Permissible grammar types are `silwordloopsil`, `ngram`, and `wordpair`.
`-fsmFName` *string*	**Required.** The filename of the output grammar WFST file. This file is in AT&T FSM format [1].
`-inSymsFName` *string*	**Required.** The filename of the input-symbols file that corresponds to the output WFST file. The entries in this file map from the numeric labels used in the FSM file to actual word strings.
`-outSymsFName` *string*	**Required.** The filename of the output-symbols file that corresponds to the output WFST file. The entries in this file map from the numeric labels used in the FSM file to actual word strings. Note that for all grammar types currently implemented, the output-symbols file will be identical to the input-symbols file.
`-genTestSeqs`	When specified, some debug sequences are output using the WFST after it is constructed, which is useful to confirm its correctness.

3.1.2 Lexicon transducer

The lexicon transducer is generated using the `lexgen` tool, which can be found in the tools directory after the source code has been successfully built. The lexicon transducer maps sequences of monophones (or more precisely, the sub-word units used in the pronunciation dictionary) to sequences of words. Any weights present in the final WFST are due to probabilities assigned to pronunciations.

The command-line help (obtained by executing `lexgen -h`) is displayed in Figure 3.2. The `lexgen` command line options are described in detail in Table 3.2.

```
#
# usage: lexgen [options]
#

Lexicon Options:
  -lexFName <string>        -> the dictionary filename - monophone transcriptions []
  -sentStartWord <string>   -> the name of the dictionary word that will start every sentence []
  -sentEndWord <string>     -> the name of the dictionary word that will end every sentence []

Context-Independent Phone Options:
  -monoListFName <string>   -> the file containing the list of monophones []
  -silMonophone <string>    -> the name of the silence phoneme []
  -pauseMonophone <string>  -> the name of the pause phoneme []
  -pauseTeeTransProb <real> -> the initial to final state transition probability of the pause model, if addPronu
nsWithEndPause is defined, then this prob becomes the pronun. prob for the base pronun (without pause at end), a
nd 1-(this value) becomes the pronun. prob for the pronun with pause at the end. [0]

Transducer Options:
  -fsmFName <string>        -> the FSM output filename []
  -inSymsFName <string>     -> the input symbols output filename []
  -outSymsFName <string>    -> the output symbols output filename []
  -addPronunsWithEndSil     -> indicates that an additional pronunciation with silence monophone at end is added
 for each word.
  -addPronunsWithEndPause   -> indicates that an additional pronunciation with pause monophone at end is added f
or each word.
```

Figure 3.2: Command line help message for the `lexgen` tool.

Table 3.2: Detailed description of `lexgen` command line options.

Option	Description
`-lexFName` *`string`*	**Required.** The name of the text file containing the pronunciation dictionary. Each line of this file is in the standard dictionary format "`word(prob) ph1 ph2 ... phN`" where the pronunciation probability, "`(prob)`", is optional. Do not specify multiple pronunciations of the same word with silence or short-pause phones at the end, as this is achieved with the `-addPronunsWithEndSil` and `-addPronunsWithEndPause` options described below.
`-sentStartWord` *`string`*	The word in the dictionary that begins every recognised sentence (e.g. `<s>`). The sentence start word is immune from the effects of the `-addPronunsWithEndSil` and `-addPronunsWithEndPause` options (i.e. no additional pronunciations are added).
`-sentEndWord` *`string`*	The word in the dictionary that ends every recognised sentence (e.g. `</s>`). The sentence end word is immune from the effects of the `-addPronunsWithEndSil` and `-addPronunsWithEndPause` options (i.e. no additional pronunciations are added).
`-monoListFName` *`string`*	**Required.** The name of a text file containing the complete list of unique phonemes (one phoneme per line) that are used in the dictionary, and elsewhere in the system (e.g. to form context dependent phone names).
`-silMonophone` *`string`*	The name of the silence phoneme, which must be present in the monophone list file. Mandatory if `-addPronunsWithEndSil` is specified, ignored otherwise.
`-pauseMonophone` *`string`*	The name of the pause phoneme, which must be present in the monophone list file. Mandatory if `-addPronunsWithEndPause` is specified, ignored otherwise.

Table 3.2: (continued)

Option	Description
`-pauseTeeTransProb` *`float`*	The (non-log) initial-to-final state transition probability for the pause model. This is ignored unless `-addPronunsWithEndPause` is specified. If a value > 0 is defined then that value is used as the probability for the "base" pronunciation (i.e. without silence or pause at the end), and $1-$ the value is used as the probability of the pronunciation with pause at the end. See Section 3.1.2 for a more detailed discussion about this.
`-fsmFName` *`string`*	**Required.** The filename of the output lexicon WFST file. This file is in AT&T FSM format.
`-inSymsFName` *`string`*	**Required.** The filename of the input-symbols file that corresponds to the output WFST file. The entries in this file map from the numeric labels used in the FSM file to the names in the monophone list file.
`-outSymsFName` *`string`*	**Required.** The filename of the output-symbols file that corresponds to the output WFST file. The entries in this file map from the numeric labels used in the FSM file to the word strings in the dictionary.
`-addPronunsWithEndSil`	Include an additional pronunciation for each word in the dictionary with the silence phoneme (specified with the `-silMonophone` option) added to the end. The sentence start word and sentence end word are not affected.
`-addPronunsWithEndPause`	Include an additional pronunciation for each word in the dictionart with the pause phoneme (specified with the `-pauseMonophone` option) added at the end. The sentence start and sentence end word are not affected. Note that the `-pauseTeeTransProb` option affects pronunciation probabilities when this flag is defined.

Dictionary file format

The dictionary/lexicon file is a text file with one entry per line. The format of each line is:

```
word(prior) ph1 ph2 ... phN
```

where `(prior)` is the optional pronunciation prior probability (non-log), and `ph1 ph2 ... phN` are the sub-word units that define the pronunciation of `word`. The sub-word units used in the dictionary are the same sub-word units in the monophone list file that is input to `lexgen` and other tools. Any amount of whitespace can be used to separate the fields.

Multiple (different) pronunciations of the same word are permitted. If pronunciation probabilities are omitted, then a probability of 1.0 is assumed, even in the multiple pronunciation case.

Do not add multiple pronunciations in the dictionary for the same word with additional silence or pause phones at the end, as this is achieved using the `-addPronunsWithEndSil` and `-addPronunsWithEndPause` options described in Table 3.2.

There is no requirement for the dictionary to be correctly sorted.

Silence handling

Silence at the start and end of utterances is accommodated by adding sentence start and end words to the dictionary (e.g. `<s>` and `</s>`) that contain only the silence monophone in their pronunciation. The sentence start and sentence end words should match any sentence markers that occur in the language model. These words are then specified to both the WFST construction tools and to the decoder using the `-sentStartWord` and `-sentEndWord` options. When using this configuration, silence at the start and end of all utterances is assumed, and the decoder output for each utterance will always begin with the sentence start word and finish with the sentence end word.

Between-word silence is allowed by defining two pronunciations for each dictionary entry. The first pronunciation is the one defined in the lexicon file. The second pronunciation is the same as the first, except with the silence monophone (as specified with the `-silMonophone` option) added to the end of the entry. The second pronunciation is not explicitly defined in the lexicon file, but is included automatically with the `-addPronunsWithEndSil` option. The sentence start and end words are not affected by the `-addPronunsWithEndSil` option. This method assumes that silence occurs only at the end of words. Appropriate part-words must be added to the dictionary if within-word silence is to be handled during decoding.

Silence is always treated as context independent. This means that there must be a HMM defined for the silence monophone. In a cross-word triphone system, the integrated transducer accepts input sequences such as: `sil sil-k+ae k-ae+t ae-t+sil sil` [1]. In the word-internal n-phone case, where the "monophones" are in fact the context dependent phonemes used in the dictionary, the integrated transducer accepts input sequences such as: `sil k+ae k-ae+t ae-t sil`.

Pause handling

Often it is desirable to model the presence of short pauses between words separately to the modelling of more general silence conditions. Similar to the handling of silence above, the transducer construction tools allow a pause monophone to be specified (using the `-pauseMonophone` option), and an additional pronunciation with pause at the end can be added automatically for each dictionary entry using the `-addPronunsWithEndPause` option with `lexgen`.

However, usually the HMM for pause (particularly when built using HTK) contains a transition from the initial non-emitting state to the final non-emitting state (HTK refers to such HMM's as *tee* models). The addition of this pause model to the end of each pronunciation models the presence of an *optional* short pause between words.

Juicer cannot decode with HMM's that contain this initial-final state transition. However, the optional pause functionality is retained firstly through the addition of the pronunciation with pause at the end as described above. The probability of the base pronunciation is then set to be the probability of the initial-final state transition in the original pause model, and the probability of the base+pause pronunciation is set to be one minus this value. Finally, when acoustic models are loaded into memory prior to decoding, any initial-final state transition is detected and removed, and the probabilities of the remaining transitions out of the initial state are renormalised.

Pause is always treated as context independent. If the pause monophone is defined, a corresponding HMM must exist. In a cross-word triphone system, pause does not break the surrounding context. This means, for example, that the integrated transducer will accept input sequences such as: "`sil sil-k+ae k-ae+t ae-t+s sp t-s+ae s-ae+t ae-t+sil sil`" (corresponding to the word sequence "`<s> cat sat </s>`" with a short pause between `cat` and `sat`).

[1]Note that in such cross-word systems where the only context independent phoneme models are silence and (perhaps) short-pause, this implies that all utterances must begin and end with silence, which further implies that the first monophone in the sentence start word pronunciation must be silence, and the final monophone in the sentence end word pronunciation must be silence.

3.1.3 Context-dependent phoneme transducer

The context-dependent phoneme (CD) transducer is generated using the `cdgen` tool, which can be found in the tools directory after the source code has been successfully built. The CD transducer maps sequences of context-dependent phonemes (i.e. sub-word units for which we have a HMM) to sequences of monophones (the sub-word units used in the pronunciation dictionary).

Both GMM-based and ANN-based acoustic models are supported. The command-line help (obtained by executing `cdgen -h`) is displayed in Figure 3.3.

GMM-based acoustic models

For GMM-based systems, acoustic models are required to be stored in a single HTK MMF (ascii) formatted file (see Chapter 7 of [7] for MMF format specification). State-level and mixture-level tying within the MMF file (using `~s` and `~m` macros) are both supported. Transition matrix tying (using `~t` is also supported. Global options can be specified using the `~o` macro. Any `~v` macros will be ignored (e.g. the `~v` macro for the variance flooring vector, usually present in MMF files produced by HTK tools, will be ignored). All other macros are not recognised by the parser, and will cause an error. Decoder input files for each utterance are HTK-format feature files.

Model tying is achieved in the same way as HTK by using a tied HMM list file. The physical model names in the tied list file must match the HMM definitions in the MMF file. The tied list file must be specified even if no model tying is used (in which case it will contain a flat list of physical model names).

Three variants of GMM-based acoustic models are supported, which impacts the way sub-word units are used in the dictionary, and the way they are specified in the monophone and tied list files.

Monophones. In this case the sub-word units in the dictionary should be monophones. The monophone list file contains a unique list of the monophones used in the dictionary (as well as the silence and pause monophones if applicable). The tied list file contains a list of the (possibly tied) monophone HMMs in the MMF file.

Word-internal (WI) *n*-phones. The sub-word units used in the dictionary should be WI *n*-phones (triphones, quinphones, ...). The monophone list file contains a unique list of the WI *n*-phones in the dictionary (as well as the silence and pause monophones if applicable). The tied list file contains a list of the (possibly tied) WI *n*-phone HMMs in the MMF file. Because context dependency does not extend across word boundaries, the WI *n*-phones can be treated as if they were monophones.

Cross-word triphones. The sub-word units used in the dictionary should be monophones. The monophone list file contains a unique list of the monophones in the dictionary (as well as the silence and pause monophones if applicable). The tied list file contains the logical and physical triphone model names. In this case the `-cdSepChars` option is required to define the characters that separate monophones in each triphone name.

ANN-based acoustic models

For hybrid HMM/ANN-based systems, the file for each utterance input by the decoder is assumed to be a LNA file containing pre-computed monophone posterior probabilities. In this case, the sub-word units in the dictionary must be monophones. The monophone list file contains the complete list of monophones, and the ordering of monophones in this file must match the ordering of the posterior probabilities in each LNA input frame. A priors file, containing the prior probability of each monophone, must also be specified and the ordering of priors must also match the ordering of the LNA frame posteriors.

The HMM topology for each monophone model is synthesised. The number of states (including non-emitting initial and final states) to be used in each monophone HMM is specified using the

```
#
# usage: cdgen [options]
#

Context−Independent Phone Options:
  −monoListFName <string>  −> the file containing the list of monophones []
  −silMonophone <string>   −> the name of the silence phoneme []
  −pauseMonophone <string> −> the name of the pause phoneme []

Context−Dependent Phone Options:
  −tiedListFName <string>  −> the file containing the CD phone tied list []
  −htkModelsFName <string> −> the file containing the acoustic models in HTK MMF format []
  −cdSepChars <string>     −> the characters that separate monophones in CD phone names (in order) []
  −cdType <string>         −> the type of context−dependency for output WFST (mono,monoann,xwrdtri) [mono]

Hybrid HMM/ANN related options:
  −priorsFName <string>    −> the file containing the phone priors []
  −statesPerModel <int>    −> the number of states in each HMM used in the hybrid system [0]

Transducer Options:
  −fsmFName <string>       −> the FSM output filename []
  −inSymsFName <string>    −> the input symbols output filename []
  −outSymsFName <string>   −> the output symbols output filename []
  −lexInSymsFName <string> −> the (pre−existing) input symbols filename for the lexicon transducer []
  −genTestSeqs             −> generates some test sequences from the CD transducer just constructed.
```

Figure 3.3: Command line help message for the `cdgen` tool.

`-statesPerModel` option. A simple left-to-right HMM is then created for each monophone with 0.5 probability self-loop transitions on each emitting state. The output likelihoods at each frame for all emitting states in the same monophone model are the same, that is the monophone posterior probability obtained from the input LNA frame scaled by the corresponding prior.

Table 3.3: Detailed description of `cdgen` command line options.

Option	Description
`-monoListFName` *string*	**Required.** The name of a text file containing the complete list of unique phonemes (one phoneme per line) that are used in the dictionary, and elsewhere in the system (e.g. to form context dependent phone names).
`-silMonophone` *string*	The name of the silence phoneme, which must also be present in the monophone list file, the tied list file model, and a model must exist in the HTK MMF file if specified.
`-pauseMonophone` *string*	The name of the pause phoneme, which must be present in the monophone list file, the tied list file, and a model must exist in the HTK MMF file if specified.
`-tiedListFName` *string*	The name of the file containing the (possibly tied) list of context-dependent phonemes. This list is in HTK HMM list format (see sect. 7.4 and fig. 7.12 in [7]), and therefore allows tying (sharing) of models between phonemes. This option is mandatory when the CD type is `mono` or `xwrdtri`.
`-htkModelsFName` *string*	The name of the HTK MMF (master macro file) containing the HMM/GMM definitions. The model names in this file must match the physical models specified in the tied list file. This option is mandatory when the CD type is `mono` or `xwrdtri`.

Table 3.3: (continued)

Option	Description
`-cdSepChars` *string*	An ordered string containing the characters used to separate monophones when constructing context-dependent phoneme names. For example, the value of this string should be "`-+`" for triphones of the form "`ax-d+ey`". This option is mandatory when the CD type is `xwrdtri` and is ignored otherwise.
`-cdType` *string*	**Required.** The type of CD transducer to be constructed. Valid types are `mono`, `monoann`, and `xwrdtri`. The `mono` type is used for a GMM-based system where the model names are the same as the sub-word units used in the pronunciation dictionary (e.g. with monophone or word-internal triphone models). The `monoann` type is used for a hybrid HMM/ANN system, where again the model names are the same as the sub-word units used in the pronunciation dictionary. The `xwrdtri` type is for GMM-based systems where the pronunciation dictionary is specified using monophones, but the HMM set has models for triphones, and the triphonic context dependency extends across word boundaries.
`-ndixt`	Enables generation of a cross-word triphone transducer that has a non-deterministic inverse. This transducer will probably have a smaller number of arcs compared to the default deterministic inverse version, potentially resulting in final integrated transducer with fewer arcs. However, the composition of this transducer with $L \circ G$ often requires prohibitive amounts of RAM. This option is ignored unless the CD type is `xwrdtri`.
`-priorsFName` *string*	The name of the file containing phone prior probabilities. This is only used for when the CD type is `monoann`. The ordering of prior probabilities must match the order of monophone names in the monophone list file. This option is mandatory when the CD type is `monoann`, and is ignored otherwise.
`-statesPerModel` *integer*	The number of states (including non-emitting initial and final states) in each HMM within a hybrid HMM/ANN system. For hybrid decoding, left-to-right HMM's for each monophone are synthesised, with a 0.5 probability self-loop transition on each emitting state. All emitting states within the same model use identical output probability distributions.
`-fsmFName` *string*	**Required.** The filename of the output CD WFST file. This file is in AT&T FSM format.
`-inSymsFName` *string*	**Required.** The filename of the input-symbols file that corresponds to the output WFST file. The entries in this file map from the numeric labels used in the FSM file to the physical model names in the tied list file.

Table 3.3: (continued)

Option	Description
`-outSymsFName` *string*	**Required.** The filename of the output-symbols file that corresponds to the output WFST file. The entries in this file map from the numeric labels used in the FSM file to the monophone names in the monophone list file.
`-lexInSymsFName` *string*	**Required.** The name of the *lexicon WFST input symbols* file. This file contains auxiliary symbols that were added to the lexicon WFST to disambiguate homophones, and these also need to be added to the CD WFST.
`-genTestSeqs`	When specified, some debug sequences are output using the WFST after it is constructed, which is useful to confirm its correctness.

3.2 Integrated WFST construction

The three transducers generated in the previous sections are combined using one of the `build-wfst`, `build-wfst-attmit` or `build-wfst-mit` scripts located in the tools directory. Note that these scripts all require the Juicer environment to be configured (i.e. the correctly configured `juicer-env` file needs to have been previously source'd).

The `build-wfst` script calls numerous command-line utilities from AT&T's FSM toolkit [3, 2] to firstly optimse the three component transducers, and then to combine them into a deterministic and minimal integrated transducer. The final integrated transducer maps sequences of context-dependent phoneme models to sequences of words that satisfy the language model constraints, and is thus ready for decoding using the Juicer decoder.

The `build-wfst-attmit` script uses the AT&T FSM toolkit for all WFST operations *except* the composition of the lexicon with the grammar transducer. This composition is done using the `fst_composelg` utility, which is a custom extension of the FST toolkit distributed by MIT. The composition algorithm used in the `fst_composelg` utility is slower than the AT&T FSM equivalent, but is more memory efficient and the composition result is deterministic.

The `build-wfst-mit` script uses the MIT FST toolkit for all WFST operations, including the custom `fst_composelg` utility described above.

The command line usage for the `build-wfst` is (same for `build-wfst-mit` and `built-wfst-attmit`):

```
build-wfst [-of] <Grammar WFST> <Lexicon WFST> <CD phoneme WFST>
```

The `-of` option enables the optimisation of the final integrated WFST (i.e. determinisation and minimisation). This optimisation is disabled by default because when the final WFST is large (e.g. in systems that use N-gram LM's), these operations often fail due to lack of RAM.

Each WFST filename must take the form *`prefix`*`.fsm`, where *prefix* includes the pathname of the file. The script also expects that the input and output symbols files corresponding to each FSM file have names *`prefix`*`.insyms` and *`prefix`*`.outsyms` respectively. All files are generated automatically by the `gramgen`, `lexgen`, and `cdgen` tools, but care must be taken so that these tools output files with names that satisfy the above requirements.

The `build-wfst` script outputs the integrated WFST to the `final.fsm` file in the same directory as the input grammar WFST. The input and output symbols are output in the same directory to `final.insyms` and `final.outsyms` respectively. Note that the input symbols file for the integrated WFST is the same as the input symbols file for the context-dependent phone transducer, and the output symbols file for the integrated WFST is the same as the output symbols file for the grammar transducer.

3.3 Decoding

The integrated WFST constructed in Section 3.2 is a phoneme-level (HMM-level) network that defines the search space of the decoder. Each path through this network is guaranteed to be a correct sequence of context dependent phonemes corresponding to a sequence of words that obeys constraints imposed by the language model. The weights in the integrated transducer are a combination of language model probabilities (scaled and offset by the LM scaling factor and word insertion penalty respectively), any pronunciation probabilities defined in the lexicon, as well as the pronunciation weights applied to implement optional short pause functionality.

The Juicer decoder dynamically expands the model-level transducer network into a state-level network that is suitable for finding the best state-level path subject to both acoustic and language model constraints. The decoding algorithm implemented within Juicer is a time-synchronous Viterbi search. Three types of pruning are supported: standard beam-search pruning at both emitting state and model-end levels, along with histogram pruning at the emitting state level. A more detailed discussion of pruning, along with WER vs. speed analysis for different pruning configurations on a 20k Wall Street Journal system is included in Appendix A.

The command-line help (obtained by executing `juicer -h`) is displayed in Figure 3.4. A detailed description of all `juicer` options is contained in Table 3.4.

Binary file creation

The HTK MMF file containing HMM definitions in a GMM-based system is often very large (e.g. > 100MB). The MMF format is also based on a flexible, but relatively complex formal specification, and `juicer` uses a grammar parser implemented with `bison` and `flex` to correctly parse the MMF format ASCII files. This parsing (along with the dynamic allocation of (many) related data structures) is a time-intensive task. The same situation applies to the FSM-format ASCII file containing the final integrated WFST, which is often hundreds of megabytes in size and takes considerable time to read from disk into appropriate data structures.

To alleviate this problem, `juicer` creates and uses binary versions of these two files, which are serialised versions of the internal data structures. The binary files are much smaller than their ASCII counterparts, and their serialised format eliminates the need for complicated parsing and large numbers of dynamic memory allocations.

When `juicer` is invoked, if binary versions of the MMF file and WFST FSM file do not exist, then their ASCII counterparts are read from disk, and binary versions are written to disk before decoding proceeds. The filenames of the binary files are the concatenation of the original ASCII filename with "`.bin`" (retaining any path information). If binary versions already exist then the ASCII versions are ignored, and the MMF HMM definitions and the WFST decoding network are read directly from the binary version.

The binary file functionality is hard-coded. This means that if the original ASCII versions of either the MMF file or the transducer FSM file are changed then any existing binary versions of these files must be removed manually, so that they are recreated at the next invocation of `juicer`.

Care also needs to be taken in a cluster computing environment. Juicer should be invoked manually on a single machine to create the binary files *before* the decoding is launched on the cluster.

```
#
# usage: juicer [options]
#

General Options:
  -logFName <string>       -> the name of the log file []
  -framesPerSec <int>      -> Number of feature vectors per second (used to output timings in recognised words, and also to calc
ulate RT factor [100]

Vocabulary Options:
  -lexFName <string>       -> the dictionary filename - monophone transcriptions []
  -sentStartWord <string>  -> the name of the dictionary word that will start every sentence []
  -sentEndWord <string>    -> the name of the dictionary word that will end every sentence []

Acoustic Model Options:
  -htkModelsFName <string> -> the file containing the acoustic models in HTK MMF format []
  -doModelsIOTest          -> tests the text and binary acoustic models load/save

Hybrid HMM/ANN related options:
  -priorsFName <string>    -> the file containing the phone priors []
  -statesPerModel <int>    -> the number of states in each HMM used in the hybrid system [0]

WFST network parameters:
  -fsmFName <string>       -> the FSM filename []
  -inSymsFName <string>    -> the input symbols (ie. CD phone names) filename []
  -outSymsFName <string>   -> the output symbols (ie. words) filename []
  -genTestSeqs             -> generates some test sequences using the network.

Decoder Options:
  -mainBeam <real>         -> the (+ve log) window used for pruning emitting state hypotheses [-3.40282e+38]
  -phoneEndBeam <real>     -> the (+ve log) window used for pruning phone-end state hypotheses [-3.40282e+38]
  -maxHyps <int>           -> Upper limit on the number of active emitting state hypotheses [0]
  -inputFName <string>     -> the file containing the list of files to be decoded []
  -inputFormat <string>    -> the format of the input files (htk,lna) []
  -outputFName <string>    -> the file where decoding results are written (stdout,stderr,<filename>). default=stdout []
  -outputFormat <string>   -> the format used for recognition results output (ref,mlf,xmlf,verbose). default=ref []
  -refFName <string>       -> the file containing word-level reference transcriptions []
  -lmScaleFactor <real>    -> the language model scaling factor [1]
  -removeSentMarks         -> removes sentence start and end markers from decoding result before outputting.
  -modelLevelOutput        -> outputs recognised models instead of recognised words.
  -latticeDir <string>     -> the directory where output lattices will be placed []
  -monoListFName <string>  -> the file containing the list of monophones []
  -silMonophone <string>   -> the name of the silence monophone []
  -pauseMonophone <string> -> the name of the pause monophone []
  -tiedListFName <string>  -> the file containing the (tied) model list []
  -cdSepChars <string>     -> the characters that separate monophones in context-dependent phone names (in order) []
```

Figure 3.4: Command line help message for the `juicer` decoder.

Table 3.4: Detailed description of `juicer` command line options.

Option	Description
`-lexFName` *string*	**Required.** The name of the text file containing the pronunciation dictionary.
`-sentStartWord` *string*	The word in the dictionary that begins every recognised sentence (e.g. `<s>`).
`-sentEndWord` *string*	The word in the dictionary that ends every recognised sentence (e.g. `</s>`).
`-htkModelsFName` *string*	The name of the HTK MMF (master macro file) containing the HMM/GMM definitions.
`-priorsFName` *string*	The name of the file containing phone prior probabilities for hybrid HMM/ANN decoding. The ordering of prior probabilities must match the order of monophone names in the monophone list file. The phone priors are used to scale the phone posterior probabilities from the input LNA file, producing scaled likelihoods that are suitable for decoding.
`-statesPerModel` *integer*	The number of states (including non-emitting initial and final states) in each HMM within a hybrid HMM/ANN system. For hybrid decoding, left-to-right HMM's for each monophone are synthesised, with a 0.5 probability self-loop transition on each emitting state. All emitting states within the same model use identical output probability distributions.

Table 3.4: (continued)

Option	Description
`-fsmFName` *string*	**Required.** The filename of the integrated WFST file. This file is in AT&T FSM format.
`-inSymsFName` *string*	**Required.** The filename of the input-symbols file that corresponds to the integrated WFST file. The entries in this file map from the numeric labels used in the FSM file to physical model names.
`-outSymsFName` *string*	**Required.** The filename of the output-symbols file that corresponds to the integrated WFST file. The entries in this file map from the numeric labels used in the FSM file to word strings in the dictionary.
`-mainBeam` *real*	The main pruning beam width (a positive log value). At each frame, a threshold is calculated by subtracting this beam width from the score of the best active emitting state hypothesis. All active emitting state hypotheses with scores less than this threshold are then deactivated. The default value of 0.0 disables this pruning.
`-phoneEndBeam` *real*	The phone-end pruning beam width (a positive log value). At each frame, and before extending hypotheses from the final states of phone models to initial states of successor phone models, a threshold is calculated by subtracting this beam width from the score of the best phone end state hypothesis. Phone end state hypotheses with scores less than this threshold are then deactivated. Typically the phone-end beam width can be smaller than the main beam width. The default value of 0.0 disables this pruning.
`-maxHyps` *integer*	Sets an upper limit on the number of active hypotheses at each frame. Has no effect if the total number of active hypotheses is less than the limit. Often used as a secondary measure in conjunction with the other pruning options, to place an upper bound on the number of active hypotheses in speech regions with high confusability. The default value of 0 disables this pruning.
`-inputFName` *string*	**Required.** A file containing the list of filenames that to be decoded. This can be a flat list of different files, or can be in "extended" HTK script file format, that allows the specification of multiple segments within a single file to be decoded.
`-inputFormat` *string*	**Required.** The format of the files that are to be decoded. For HMM/GMM decoding, input files must be in HTK feature file format (see Section 5.10.1 of [7]), and this value must be `htk`. For hybrid HMM/ANN decoding, input files must be in LNA 8-bit format (see Appendix B), and this value must be set to `lna`.
`-outputFName` *string*	The name of the file where the decoding output is placed. This can be a filename, `stdout`, or `stderr`. Default is `stdout`.

Table 3.4: (continued)

Option	Description
`-outputFormat` *string*	The format of the decoder output. Valid values are `ref` (all words from one utterance/file per line), `mlf` (HTK MLF format, words only, no timings), `xmlf` (HTK MLF format, words+timings), or `verbose` (nice formatted results, use with `-refFName` option). See Section 6.3 of [7] for MLF details. Note that due to the WFST framework, word start/end times will be incorrect. Model start/end times obtained by using the `xmlf` output format in conjunction with the `-modelLevelOutput` flag will be correct.
`-refFName` *string*	A file containing reference transcriptions. This option is only useful in combination with the `verbose` output format to have reference and recognised words displayed after each file is decoded.
`-lmScaleFactor` *real*	The log weights in the integrated WFST are multiplied by this value before they are used during decoding. Care must be taken, because the integrated WFST may already have LM scaling and word insertion penalty applied. This option will blindly scale all weights.
`-removeSentMarks`	Removes the sentence start and end words from the output decoding result. Use is only advised when the output format is `verbose`.
`-modelLevelOutput`	Activates model level output. Physical model names (as opposed to logical names) are output, so care must be taken when interpreting results if model tying is being utilised. If this flag is specified, then information about monophones and context-dependent phones must be provided using the `-monoListFName`, `-silMonophone`, `-pauseMonophone`, `-tiedListFName`, and `-cdSepChars` options.
`-monoListFName` *string*	The name of a text file containing the complete list of unique phonemes (one phoneme per line) that are used in the dictionary, and elsewhere in the system (e.g. to form context dependent phone names). Used in conjunction with `-modelLevelOutput`. Also required for hybrid HMM/ANN decoding.
`-silMonophone` *string*	The name of the silence phoneme. Only used in conjunction with `-modelLevelOutput`.
`-pauseMonophone` *string*	The name of the pause phoneme. Only used in conjunction with `-modelLevelOutput`.
`-tiedListFName` *string*	The name of the file containing the (possibly tied) list of context-dependent phonemes. This list is in HTK HMM list format (see sect. 7.4 and fig. 7.12 in [7]), and therefore allows tying (sharing) of models between phonemes. Only used in conjunction with `-modelLevelOutput`.
`-cdSepChars` *string*	An ordered string containing the characters used to separate monophones when constructing context-dependent phoneme names. For example, the value of this string should be “`-+`” for triphones of the form “`ax-d+ey`”. Only used in conjunction with `-modelLevelOutput`.

Table 3.4: (continued)

Option	Description
`-latticeDir` *string*	If specified, then lattice generation is enabled. Defines the directory where output lattice files are written. For now, the lattices are output as WFST's in FSM format.

Chapter 4

Examples

A number of example decoding configurations have been provided to demonstrate how to decode with the various types of acoustic and language models supported by Juicer.

The following examples are provided:

`numbers_gmm_mono`. OGI numbers, GMM-based, Monophone HMM's, Simple word-loop grammar.

`numbers_gmm_wrdi`. OGI numbers, GMM-based, Word-internal triphone HMM's, Simple word-loop grammar.

`numbers_gmm_xwrd`. OGI numbers, GMM-based, Cross-word triphone HMM's, Simple word-loop grammar.

`numbers_ann_mono`. OGI numbers, hybrid HMM/ANN-based, Monophone HMM's, Simple word-loop grammar.

`rm_gmm_xwrd`. Resource management, GMM-based, Cross-word triphone HMM's, Word pair grammar.

There are two scripts provided for each example system. The `make-wfst` script builds the integrated transducer network, and the `do-decode` runs the decoder. By default the `make-wfst` script calls the `build-wfst` script to build the integrated transducer. Calling "`make-wfst -attmit`" or "`make-wfst -mit`" forces the `build-wfst-attmit` and `build-wfst-mit` respectively to be called instead of the default `build-wfst`.

4.1 Obtaining the examples

- Copy.
  1. Copy the examples `.tar.gz` file to the top-level directory created in Chapter 2 (i.e. `<dir>`).
  2. `cp /home/speech/moore/juicer_rel/juicer-examples-0.5.0.tar.gz <dir>`
- Extract files.
  1. `cd <dir>`
  2. `tar -xzvf juicer-examples-0.5.0.tar.gz`
  3. The `<dir>/examples/` directory should now contain subdirectories for each example system.
- Setup paths in example files.
  1. `cd <dir>/examples`
  2. `./fix-paths`

Chapter 5

Current Limitations

Memory requirements

When N-gram back-off language models are used, the grammar transducer contains an arc for every n-gram weight and an arc for each back-off weight specified in the ARPA LM file. In addition, back-off arcs all have ϵ labels, which increases the complexity of composition with the lexicon WFST and the size of resulting composed WFST. In cases where the N-gram grammar transducer is huge, the AT&T tools require large amounts of memory to produce the final integrated transducer, and often the composition or the subsequent determinisation will fail due to lack of memory.

The final integrated transducer can also be very large, because it is an explicit representation of the entire search space. AT&T often (e.g. [4]) use N-gram back-off language models that have been pre-shrunk using, for example, the method of Seymore and Rosenfeld [5] in order to reduce the size of the grammar WFST, preventing failure of composition/determinisation due to lack of memory, and resulting in a final transducer of manageable size.

The inability of AT&T tools to handle large N-gram LM systems is currently the greatest weakness of Juicer, and is the sole factor preventing its use on tasks with large $\geq$3-gram LM's. A future release will hopefully address this problem using, for example, on-the-fly (instead of the current offline) composition with higher-order N-gram LM WFST's, or by relying on WFST's at the phoneme and lexical levels, and incorporating LM knowledge using more conventional decoding techniques.

Note that when the final integrated transducer is large, the `juicer` decoder also requires a large amount of memory (usually much more than other decoders) because it must load the entire integrated transducer into a suitable data structure.

Lack of some HTK functionality

Some of the non-core functionality that is present in HTK has not yet been implemented within `juicer`. However, in all cases HTK tools can be easily applied offline to overcome any limitations. Missing functionality includes:

On-the-fly dynamic feature calculation. HTK has the ability to calculate first- and second-order derivatives on-the-fly so that only base features need to be stored on hard disk. Juicer cannot currently do this. Input feature files to `juicer` must contain all feature vector elements, including all derivatives.

On-the-fly feature transformation. HTK has the ability to apply a transformation matrix to input feature vectors on-the-fly. Juicer cannot currently do this. Any feature transformation must be done offline, and transformed features stored in files that can be input to `juicer`.

On-the-fly model transformation. HTK has the ability to apply model transformations (e.g. those calculated using MLLR) when models are loaded at run-time. Juicer cannot do currently do this. Any model transformations must be applied off-line and the transformed models must be stored in a separate file.

Forced alignment. The HTK HVite tool has the ability to input a reference transcription along with the input feature vector stream and align the input features with the words (or phonemes) in the transcription. Juicer cannot currently do this.

Appendix A

Pruning

A time-synchronous Viterbi-based search through a network involves updating a partial hypothesis for each state of the network at each time instant. A partial hypothesis associated with a particular state in the network represents the best path through the network that ends in that state at that time instant. A simplistic Viterbi search implementation would require the partial hypothesis in every state of the network to be updated for each input speech frame (e.g. 100 times per second). This is feasible for small speech recognition tasks, such as a digits recognition task with a simple word-loop grammar. However, the networks used in large vocabulary tasks with more sophisticated language models can easily contain tens of millions of states, and the straightforward Viterbi implementation is completely unsuitable in terms of the amounts of computing power and memory required.

As a result, speech decoders are usually implemented with one or more types of hypothesis pruning. The objective of pruning is to remove all but the most likely hypotheses during the search, significantly reducing the overall search effort.

In time-synchronous decoders the most common type of pruning used is *beam-search* pruning. At each time instant the most-likely partial hypothesis (the partial hypothesis with the "best" (lowest cost) accumulated score) is determined. A threshold is then calculated by subtracting a constant log value (the *beam-width*) from the best accumulated log score. Partial hypotheses with accumulated scores below this threshold are removed from further consideration. Beam-search pruning is "adaptive" in the sense that if all partial hypotheses have scores that are very close to the best score (i.e. there is a high degree of confusability), then they are all retained. If only a handful of partial hypotheses are promising, then only those few are retained.

Juicer has two levels of beam-search pruning. The first is a global pruning as described above. The second level only operates on hypotheses that are at the end of phonemes. The pruning threshold in this case is based on the best phoneme end hypothesis rather than the global best. The beam-width used for phoneme end pruning is usually narrower than the global beam-width, and thus only the most promising phoneme end hypotheses undergo the relatively expensive operation of extension to successor phonemes.

A variant of beam-search pruning is *histogram pruning* [6], where a histogram of partial hypothesis scores is used at each time instant to determine the beam-width that will result in at most N hypotheses being retained. This method is relatively inflexible because it imposes a hard limit on the number of active hypotheses, regardless of the amount of confusion present in the network. Therefore, the two levels of beam-search pruning described previously normally serve as the primary pruning mechanism, with histogram pruning employed at a more relaxed level to prevent explosions in the search space during periods of high confusion.

Beam-search and histogram pruning are both *non-admissable*, because it is possible that a partial hypothesis destined to be on the globally best path will be pruned at an intermediate stage during decoding. There is thus a trade-off between the degree of pruning (and therefore the speed of decoding), and the recognition accuracy. In practice, large decoding speed-ups can be achieved with negligible effect on recognition accuracy. This is demonstrated on a large vocabulary task in the following section.

A.1 WSJ 20k pruning analysis

The effect of pruning on recognition speed and accuracy was assessed on a WSJ 20k task. Acoustic models were phonetic decision tree state clustered triphone models trained on the `si_tr_s` set (38,275 utterances). The language model was a modified version of the standard open vocabulary, verbalised punctuation, backoff bigram LM produced by MIT and distributed with the WSJ1 corpus. The original word list (20,002 words) was reduced to 18,453 words, removing all words for which pronunciations were not readily available. The original bigram LM was then re-normalised to match the reduced word list using the HTK `LNorm` tool.

The test set was the `si_dt_20` standard 20k development test set from the WSJ1 corpus, consisting of 503 utterances. The tests were run in a parallelised manner on a 16-CPU linux cluster. Each machine on the cluster was a dual AMD Athlon MP 2800+ (i.e. total of 8 dual-CPU machines).

Table A.1 shows the sizes of the intermediate and final WFST's.

WFST	**States**	**Arcs**
G	18,454	1,341,223
L	442,634	499,176
C	5,899	209,678
$det(L \circ G)$	1,026,044	2,854,292
$det(C \circ det(L \circ G))$	1,317,235	5,408,694

Table A.1: WFST sizes for WSJ 20k pruning analysis. G = grammar, L = lexicon, C = CD-phoneme, $\circ$ = composition operation, det = determinisation operation.

The three types of pruning implemented within `juicer` were firstly analysed in isolation, and then their combined effect was assessed. Each figure provided in the following sections plots word recognition accuracy (%) vs. decoding speed (times real-time). The "baseline" word-accuracy achieved with minimal pruning was 81.90%. On each graph two horizontal lines mark the word accuracy levels that are 1% and 2% absolute below this baseline.

Global beam-search pruning

This type of pruning is controlled using the `-mainBeam` option to `juicer`. Figure A.1 shows the effect of varying the global beam-width from 250.0 (light pruning) to 120.0 (heavy pruning). By narrowing the pruning beam-width, large speed gains are achieved with only a small effect on recognition accuracy (from 40xRT to $\sim$ 9xRT with 1% loss in accuracy).

Phone-end beam-search pruning

This type of pruning is controlled using the `-phoneEndBeam` option to `juicer`. Figure A.2 shows the effect of varying the phone-end pruning between 90.0 and 250.0 both in isolation, and also when used in conjuction with a fixed (and relatively broad) main-beam of 250.0.

The main observation from Figure A.2 is that phone-end pruning is ineffectual when used in isolation. However, when phone-end pruning is coupled with a relatively low level of global beam pruning (250.0), a better speed/accuracy trade-off is achieved than using global pruning in isolation, with 1% accuracy loss achieved at $\sim$ 5xRT.

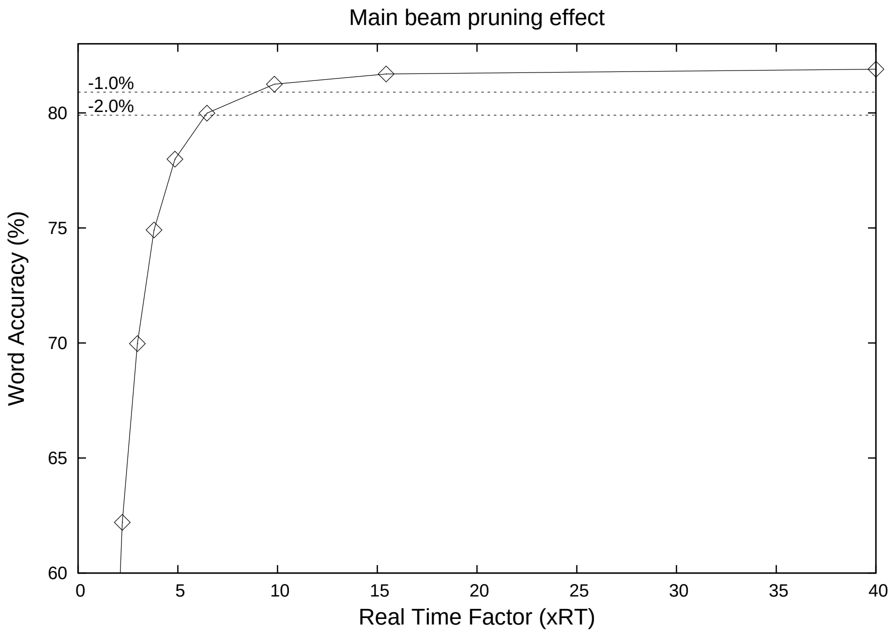

Figure A.1: Main-beam pruning effect.

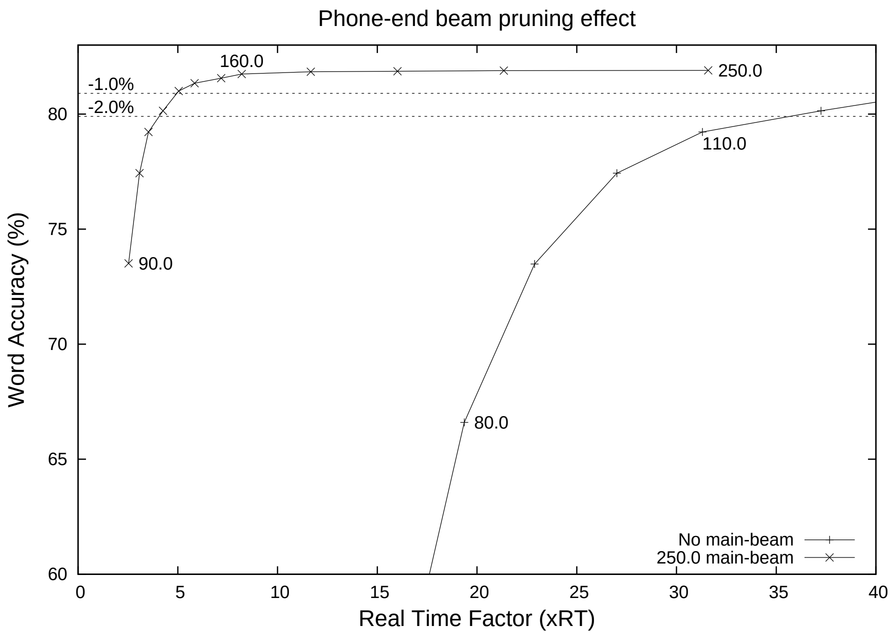

Figure A.2: Phone-end pruning effect.

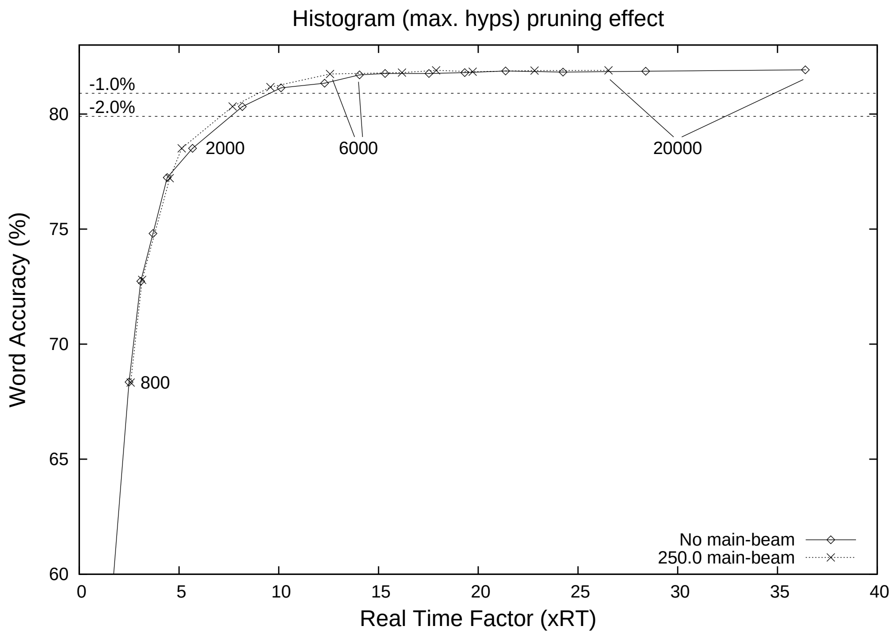

Figure A.3: Histogram pruning effect.

Histogram (maximum hypotheses) pruning

This type of pruning is controlled using the `-maxHyps` option to `juicer`. Figure A.3 shows the effect of varying the histogram pruning between 800 and 20000 both in isolation, and also when used in conjuction with a fixed (and relatively broad) global pruning beam of 250.0.

Histogram pruning in isolation has a very similar speed/accuracy tradeoff to global-beam pruning. Combining histogram pruning with light global-beam pruning results in a slightly improved speed/accuracy tradeoff for real-time factors between 5 and 15, where the histogram pruning imposes an upper limit on the number of active hypotheses that remain after global pruning. In the region below 5xRT, the tight histogram pruning dominates, and therefore no change is observed when histogram pruning is augmented with light global pruning.

Combined pruning

Figures A.4 and A.5 show decoding performance over a range of phone-end and histogram pruning values, for global-beam pruning values of 250.0 and 200.0 respectively. Each line in the two graphs corresponds to varying the histogram pruning factor over the range $[1000, 20000]$ with a particular combination of global and phone-end pruning.

The main observation from figures A.4 and A.5 is that the two levels of beam search pruning (global and phone-end) are the most important for attaining a desired speed/accuracy tradeoff. The effect of histogram pruning is to slightly improve upon the tradeoff already achieved with a given combination of global and phone-end pruning, and the improvement is relatively independent of the beam-widths used. For example, by following each line starting from the lower left, it can be seen that the optimal histogram pruning threshold occurs at roughly the fourth point (i.e. at a threshold of 6000) regardless of the global and phone-end beam-widths.

The best decoding speeds achieved in these experiments were $\sim$ 3.9xRT with 1% accuracy loss and $\sim$ 2.7xRT with a 2% accuracy loss. However, the combinations of the 3 pruning factors tested were far from exhaustive, and the trends in the graphs indicate that slightly better tradeoffs exist.

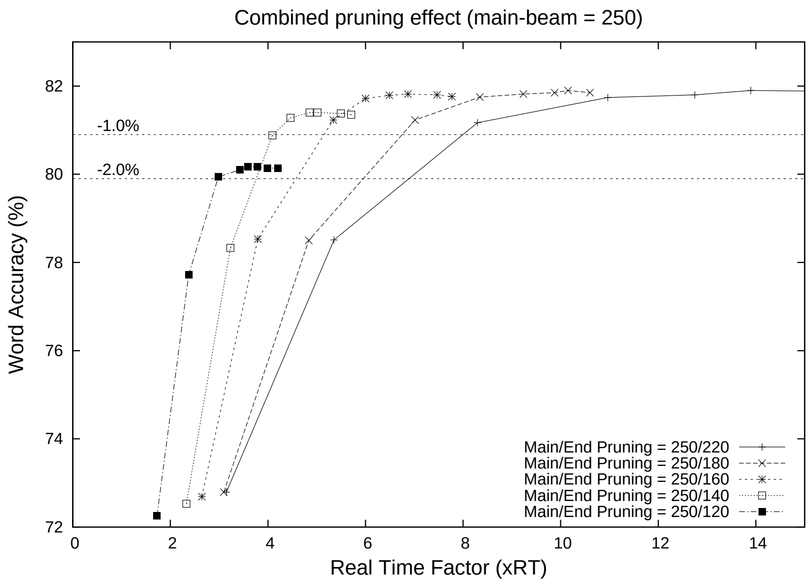

Figure A.4: Combined pruning effect. Global beam-width = 250.0. Each line obtained by varying histogram pruning from 1000 to 20000 with a fixed phone-end beam-width.

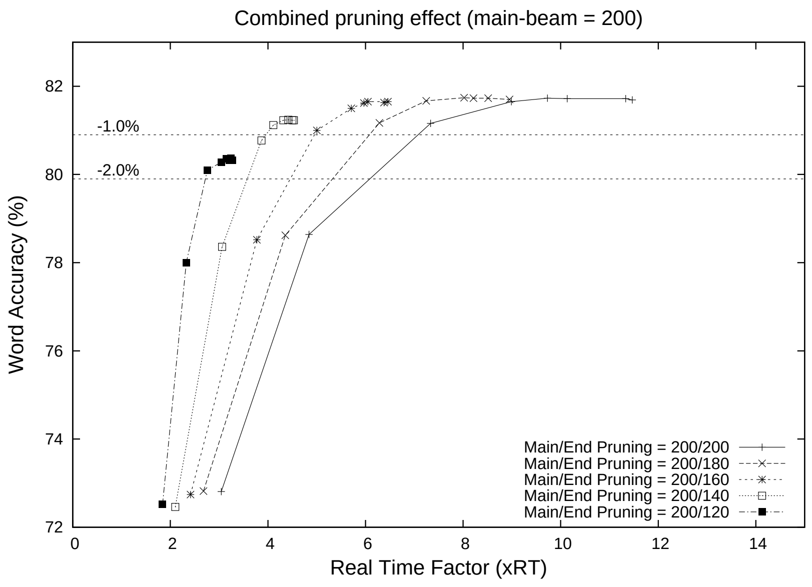

Figure A.5: Combined pruning effect. Global beam-width = 200.0. Each line obtained by varying histogram pruning from 1000 to 20000 with a fixed phone-end beam-width.

Conclusion

Pruning is a vital tool for efficiently limiting the search space in large vocabulary decoding. Significant speedups can be achieved with minimal loss in accuracy. Juicer implements three types of pruning: two-levels of beam-search pruning (global and phone-end) and histogram pruning. The two beam-search pruning types are the most important for achieving a desired tradeoff between speed and loss of accuracy. Histogram pruning can provide a further speedup without significantly affecting accuracy, and the optimal value is relatively independent of the beam-search pruning parameters.

A good strategy for finding the optimal pruning parameters could be to :

1. Run a test with light global-beam pruning only to discover the best accuracy that can be realistically achieved.
2. Decide upon an accuracy cut-off that represents an acceptable trade-off for fast decoding.
3. Experiment with tighter global pruning beam-widths, until accuracy cut-off is approached.
4. Add phone-end pruning, starting with a value the same as the global beam-width
5. Tighten phone-end pruning beam-width, until accuracy decreases to cut-off value.
6. Add histogram pruning, starting with a large value.
7. Decrease histogram pruning threshold until accuracy drops below desired cut-off.

Appendix B

LNA File Format

NAME

lna − compressed format for MLP output probablility files

SYNOPSIS

***.lna**

DESCRIPTION

lna is a compression format for speech developed by Tony Robinson, used by **y0**(1) and **noway**(1). There are really two *lna* formats (8 bit and 16 bit) supported by the software, but everybody just uses 8 bit.

Basically, each floating point probability is quantized to an 8 or 16 bit integer by the following formula:

intval = floor(−**LNPROB_FLOAT2INT** * log(*x* + **VERY_SMALL**))

where **LNPROB_FLOAT2INT** is 24 for 8 bit, and 5120 for 16 bit. The int is then pinned to between 0 and 255 (or 65535). **VERY_SMALL** prevents ugliness if the probability is 0.0.

As for the actual file format, it is a binary stream of frames, where each frame consistes of a fixed number of 8 or 16 bit values.

EOS Val0 Val1 Val2 ... Valn

EOS is 0x80 if the frame is the last frame in a sentence, 0 otherwise. *Val0* ... *Valn* are the quantized integers corresponding to the probabilities.

SEE ALSO

lna2y0new(1), **rap2lna**(1)

AUTHOR

This man page was written by:

Jonathan Segal <jsegal@ICSI.Berkeley.EDU>
Eric Fosler <fosler@ICSI.Berkeley.EDU>.

updated by: Alfred Hauenstein <alfredh@icsi.berkeley.edu>

Appendix C

BSD License

Bibliography

[1] C. Allauzen, M. Mohri, F. Pereira, and M. Riley. AT&T FSM library - file formats manual page. `http://www.research.att.com/projects/mohri/fsm/doc4/fsm.5.html`.

[2] M. Mohri, F. Pereira, and M. Riley. The design principles of a weighted finite-state transducer library. *Theoretical Computer Science*, 231(1):17–32, 2000.

[3] M. Mohri, F. Pereira, M. Riley, and C. Allauzen. AT&T FSM library - finite state machine library. `http://www.research.att.com/sw/tools/fsm`.

[4] M. Mohri, M. Riley, D. Hindle, A. Ljolje, and F. Pereira. Full expansion of context-dependent networks in large vocabulary speech recognition. In *Proceedings of ICASSP '98*, 1998.

[5] K. Seymore and R. Rosenfeld. Scalable backoff language models. In *Proceedings of ICSLP '96*, pages 232–235, 1996.

[6] V. Steinbiss, B.-H. Tran, and H. Ney. Improvements in beam search. In *Proceedings of ICSLP '94*, pages 2143–2146, 1994.

[7] S. Young, G. Evermann, T. Hain, D. Kershaw, G. Moore, J. Odell, D. Ollason, D. Povey, V. Valtchev, and P.Woodland. *The HTK Book (for HTK Version 3.2.1)*. Cambridge University Engineering Department, 2002.