Dynamic Programming: A Generic Viterbi Algorithm

Brook Milligan

Department of Biology
New Mexico State University
Las Cruces, New Mexico 88003
brook@nmsu.edu

BoostCon 2009
7 May 2009

Goals

- Introduce a generic dynamic programming library based upon the Viterbi algorithm
- Recall the Viterbi algorithm
- Abstract the generic concepts of the Viterbi algorithm
- Formally define the generic concepts
- Illustrate applications of the library
- Gather feedback on the design and utility of the library
- Encourage Boost to support scientific programming libraries

Outline

- Motivation: sequence alignment and edit distance problems
- The Viterbi algorithm: its origin in communications
- Designing a generic library: generalizing the Viterbi algorithm
- Applications: using the generic library
- Conclusion: accomplishments, questions, future, and potential

Sequence Alignment and Related Problems

Pairwise sequence alignment

Given two sequences of symbols, X and Y, a set of operations O that can transform one sequence into another, and a score $s \in S$ associated with each operation $o \in O$, what is the optimal (with respect to S) sequence of operations that transforms X into Y?

Applications

- String editing
- Molecular sequence alignment
- Sychronization of time series
- Detection of change in a sequence

Mathematics of Sequence Alignment

Let $D(i,j)$ be the score of an alignment between sequences X and Y up until the positions x_i and y_j in each. The matrix D is constructed recursively as:

$$D(i,j) = \max \begin{cases} D(i-1,j-1) + s(x_i, y_j) \\ D(i-1,j) + g \\ D(i,j-1) + g \end{cases} \tag{1}$$

where $s(x_i, y_j)$ is the score associated with synchronizing the sequences at position (i,j) and substituting y_j for x_i, and g is the score associated with not sychronizing the sequences. The initial condition is generally $D(0,0) = 0$.

Simple Example

Consider the alignment of the two sequences SEND and AND.
Let the scoring system be:

$$D(i,j) = \begin{cases} 1 & \text{if } x_i = y_j \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

$$g = -1 \quad (3)$$

Some possible alignments are:

```
SEND
-AND   Score:   1
A-ND   Score:   1
AN-D   Score:   0
AND-   Score:  -1
```

Aligning SEND and AND

		S	E	N	D
	0	-1	-2	-3	-4
A	-1	0	-1	-2	-3
N	-2	-1	-1	0	-1
D	-3	-2	-1	-1	1

SEND
-AND

Performance

Naive implementation

- Time: $O(mn)$ — Require solutions for all partial alignments
- Space: $O(mn)$ — Must store entire set of solutions

Simple optimization

- Time: $O(mn)$ — Require solutions for all partial alignments
- Space
  - $O(\min(m, n))$ — Only require immediately preceding solutions for total score
  - $O(m + n)$ — Must store complete alignment

Optimizations are only possible after careful analysis of the structure of the recursions involved in defining $D(i, j)$.

Sequence Alignment and Dynamic Programming

Properties of the alignment method

- The optimal alignment of the two entire sequences X and Y is composed of the optimal alignment of pairs of subsequences terminating at positions $x_i \forall i$ and $y_j \forall j$
- Only one optimal alignment terminates at position (x_i, y_j)
- Hence, only one score and (optionally) one pathway need be retained for each position (x_i, y_j)

The alignment method systematically evaluates potential subsequence alignments starting with the shortest possible and progressing until both entire sequences are aligned.

This is a classical dynamic programming approach.

Dynamic Programming

General properties

- Optimal substructure
  - The optimal solution to a problem is composed of optimal solutions to subproblems.
- Overlapping subproblems
  - Solutions to subproblems may be used many times in the composition of solutions to larger problems.

Viterbi algorithm

- Yields the optimal solution to dynamic programming problems
- Markov property: requires that the score for a particular subproblem depends only on:
  - Optimal score of the next smallest subproblem, and
  - Score of extending that solution into a potential solution of the current subproblem

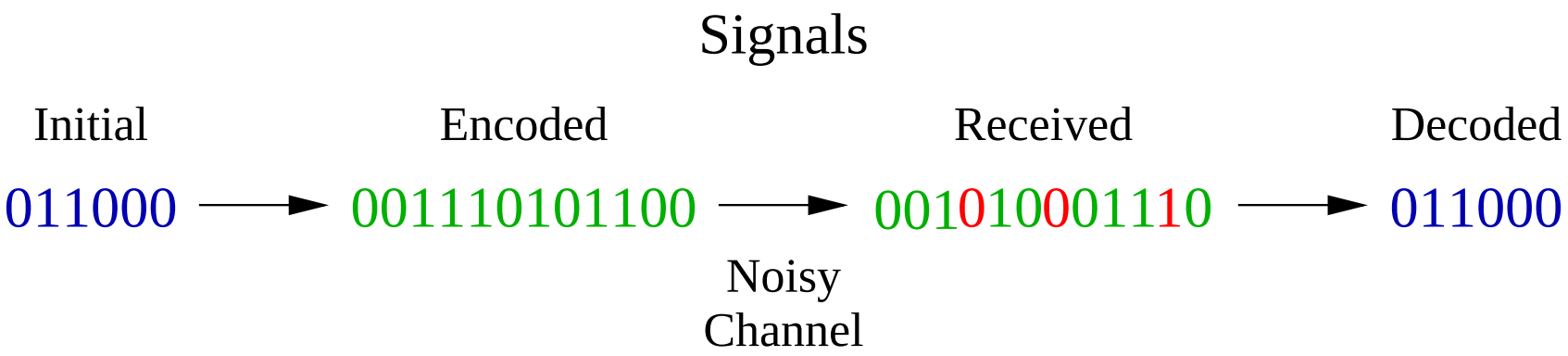
The Viterbi Algorithm

Origin

- Andrew J. Viterbi. 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm, IEEE Transactions on Information Theory 13(2): 260–269.
- Invented to decode communications signals in the presence of noise
- Continues to be the most widespread use
  - Satellite communications
  - Cell phones
  - 802.11 wireless signaling

Applications of the Viterbi algorithm to communications provides the clearest motivation for a generic Viterbi algorithm.

The Viterbi Algorithm in Communications

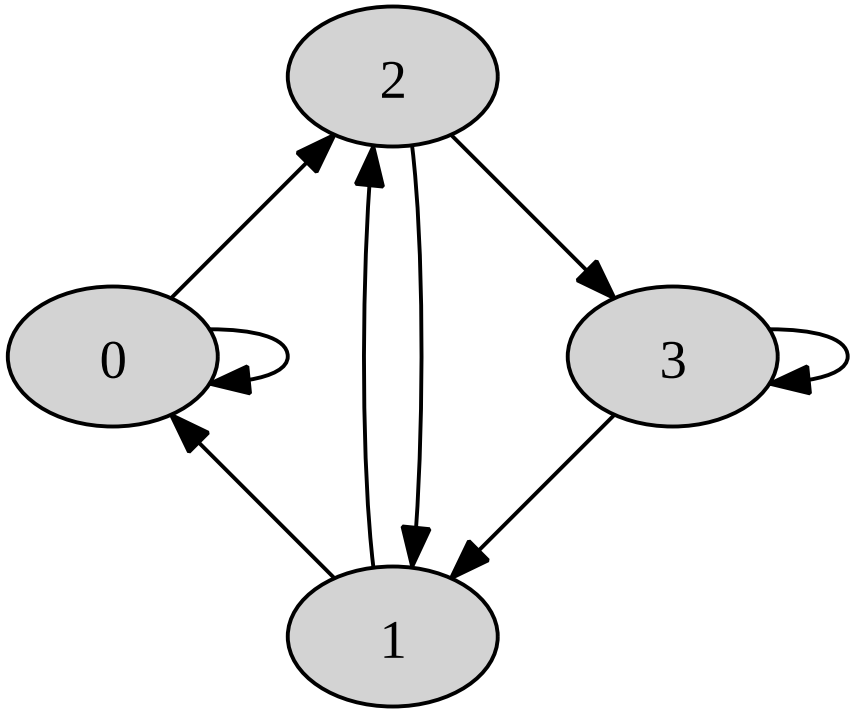


Problem statement

- Given a sequence of symbols
- Transmitted through a noisy communication channel
- Find the sequence of symbols corresponding to the optimal reconstruction of the original sequence

The Viterbi Algorithm in Communications

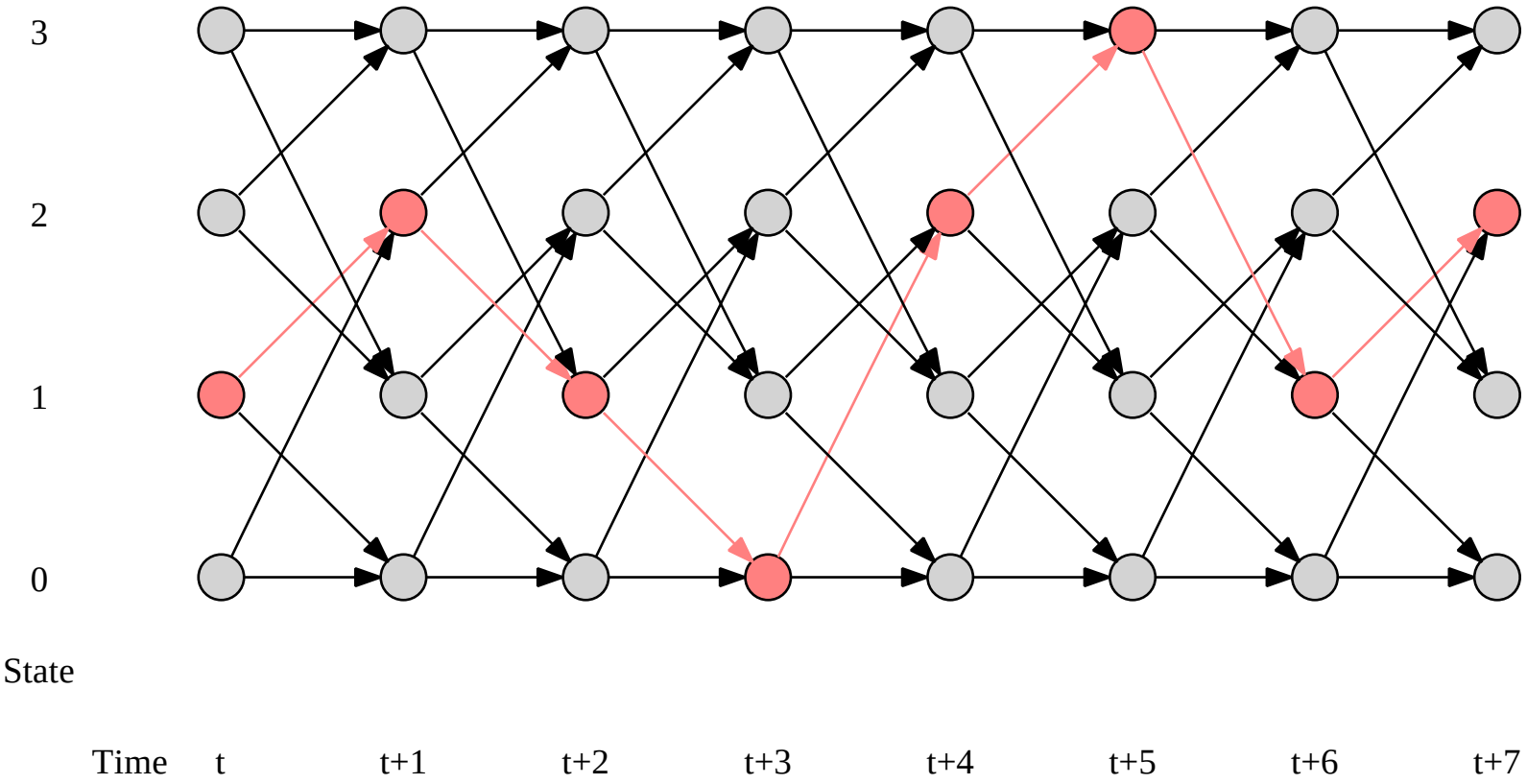
001010001110



Components

- Input symbol sequence
- Finite state machine
- Current score: match between current state and current symbol(s)
- Sequence score: accumulated score of decoded sequence
- Optimal decoded sequence

The Viterbi Algorithm and Trellis Graphs

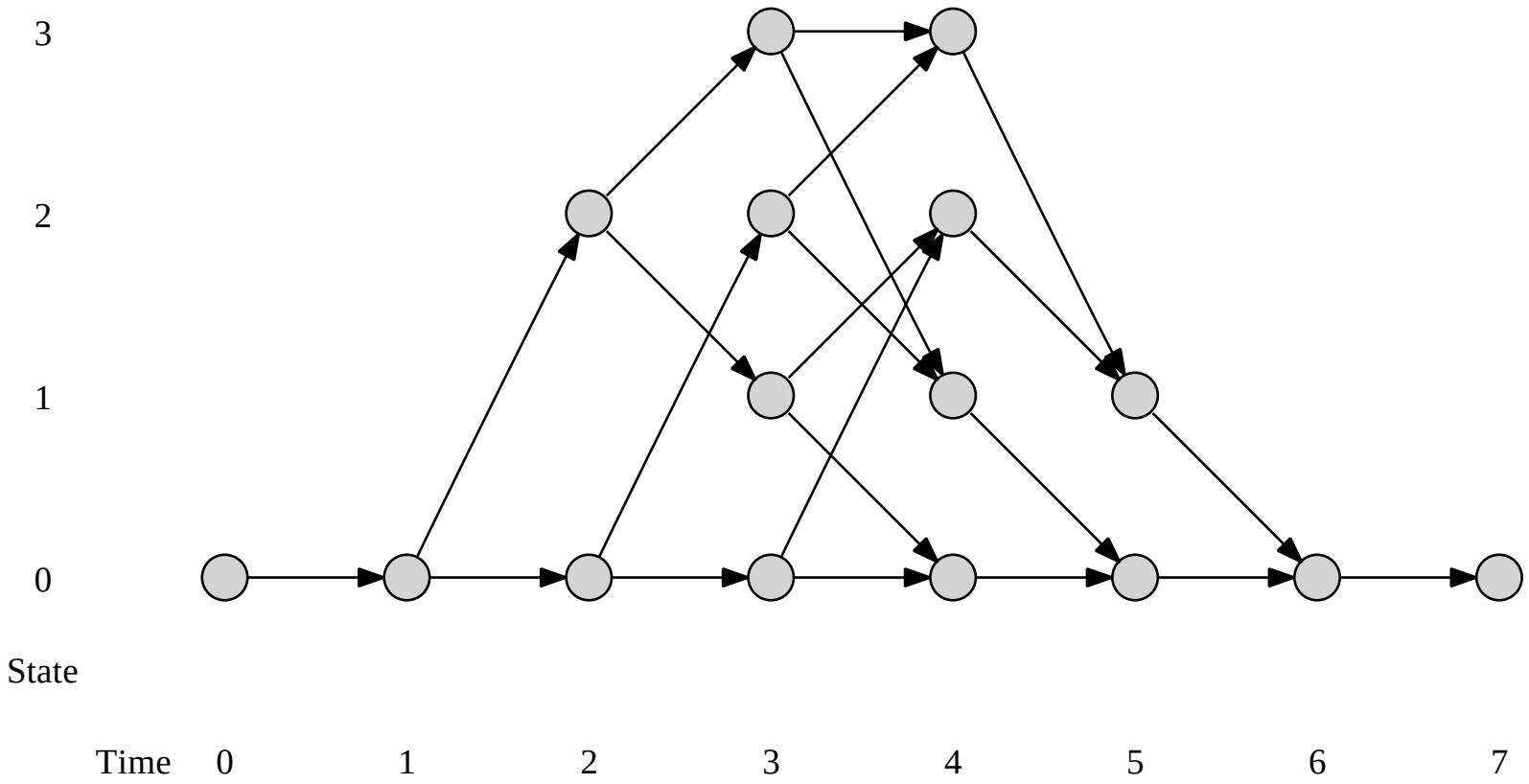


The Trellis Graph in Practice

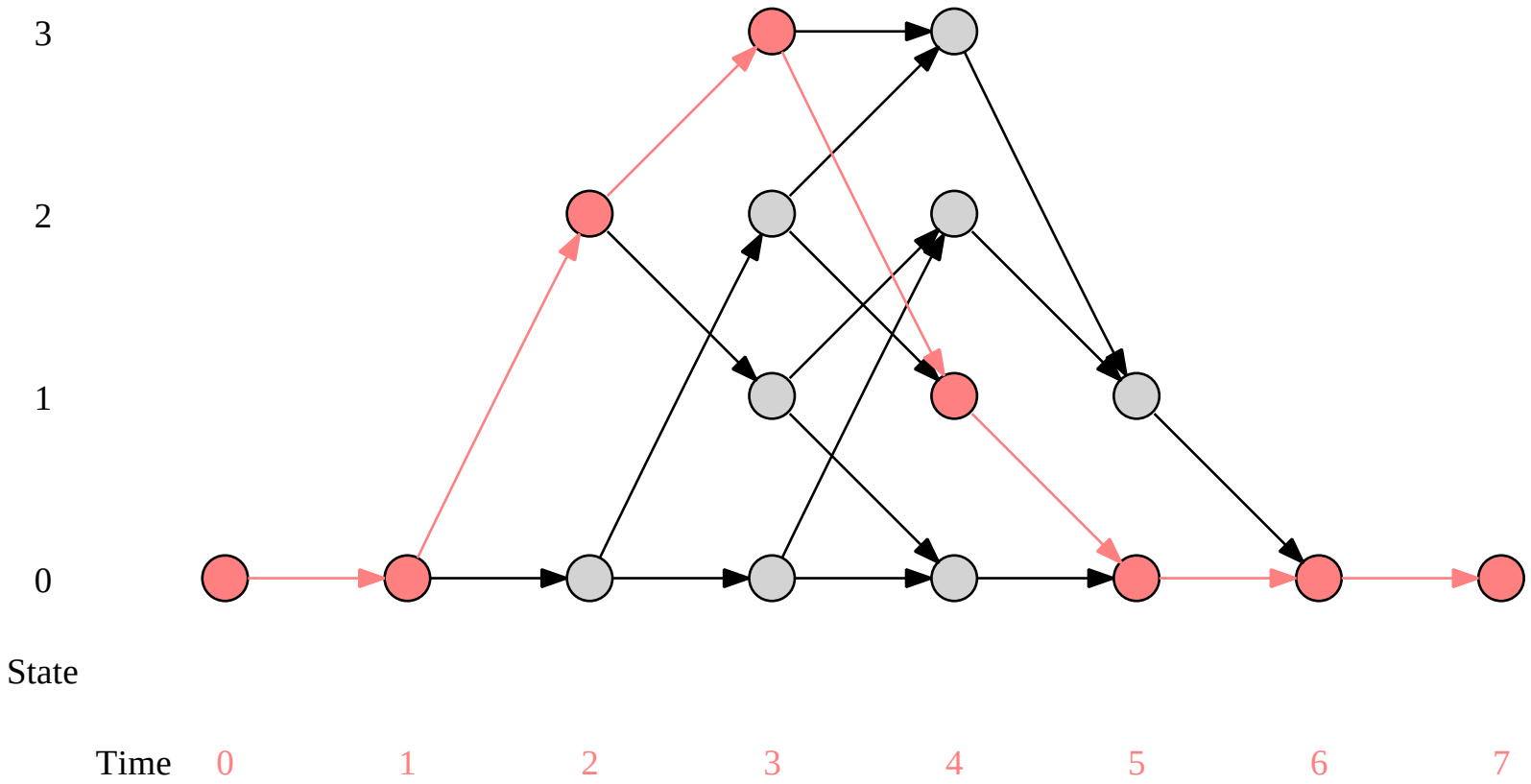
Characteristics of practical trellis graphs

- A single initial node
  - Represents the state prior to any input symbols
- A single final node
  - Represents the state after all input symbols
- Restrictions on the states corresponding to the first and last symbols
  - Many convolution codes require intial and final states of 0

Trellis Graph for a Short Message



Viterbi Decoding a Short Message



The Viterbi Algorithm and Trellis Graphs

General characteristics

- Potential paths extended with each new symbol
- Only one surviving alternative pathway retained for each vertex
- Prematurely terminated, i.e., "dead," pathways discarded
- Partial solutions available prior to completion

Designing Generic Requirements for a Viterbi Algorithm

Required components

- Optimal path algorithm
- Policy for controlling the optimal path algorithm
- Property maps
  - Score of each graph vertex
  - Predecessor vertices along the optimal path

Available components

- Boost Graph Library (BGL)
  - Clearly defined Graph Concepts
  - Several optimal path algorithms
- Boost Property Map Library
  - Clearly defined Property Map Concepts
  - Several useful property map wrapper classes
  - Interfaces well with the BGL

Boost Graph Library

Optimal path algorithms for directed graphs

	Graph Concepts		
Algorithm	Vertex List	Edge List	Incidence
Bellman-Ford		×	
Dijkstra	×		×
Directed acyclic graph	×		
Floyd-Warshall all pairs	×	×	
Johnson all pairs	×	×	×

Vertex List Graph Concept Range across all vertices

Edge List Graph Concept Range across all edges

Incidence Graph Concept Range across the out edges of any given vertex

Trellis Graphs and the BGL Algorithms

Concept correspondence

- Local graph properties are always known for trellis graphs
  - Adjacent vertices: Adjacency Graph
  - Out edges: Incidence Graph

The set of vertices may be unknown until the signal is complete.

- Global graph properties need not be known, but are required by all BGL algorithms
  - List of vertices: Vertex List Graph
  - List of edges: Edge List Graph

BGL optimal path algorithms cannot be used directly for a Viterbi algorithm.

Property Map Requirements for the Viterbi Algorithm

Size of a trellis

The set of vertices in a trellis graph may be huge. Consider the following:

- Rate $= \frac{1}{2}, k = 7$: A common convolution code used by NASA in satellite communication
  - For every bit in the message, two bits are sent
  - There exist $2^7 = 128$ internal states of the encoder/decoder
- Symbols are transmitted at a rate of 1 GHz
- The trellis graph accumulates $1.28 \times 10^{11} \frac{\text{vertices}}{\text{second}}$

Property Map Requirements for the Viterbi Algorithm

Implications for property maps

The set of vertices and hence property maps in a trellis graph may be huge and unknown at the outset. However,

- Dead pathways can be trimmed from the graph.
- Partial solutions can be emitted prior to algorithm completion.

This requires property maps that can be modified while the algorithm proceeds.
But how can we reliably recognize truly dead pathways and partial solutions?

Properties of Trellis Graphs

A graph $G = (V, E)$ is a trellis graph if

- Each vertex $v \in V$ is characterized by two properties: `time(v)` and `state(v)`.
- Each edge $e = (u, v) \in E$ is directed, i.e., (u, v) is an ordered pair
- A single source vertex exists, i.e., exactly one vertex $v \in V$ has no in edges
- A single target vertex exists, i.e., exactly one vertex $v \in V$ has no out edges
- No cycles exist
- The graph is weakly connected
  - For every pair of vertices $u, v \in V$, there exists a path (not necessarily always following the direction of edges) between them.

Properties of Trellis Graphs (continued)

A graph $G = (V, E)$ is a trellis graph if

- Edges are strictly monotonically increasing in time, i.e., for every edge $e = (u, v) \in E$

$$\text{time}(u) < \text{time}(v) \tag{4}$$

Monotonicity and Property Maps

- A vertex v is finished when all of its out edges, and hence adjacent vertices, have been evaluated.
- All vertices adjacent to v have a time strictly greater than for v itself.
- Therefore, when v is finished all vertices u with $time(u) < time(v)$ have also been finished.
- An earlier vertex u is
  - Dead if there is no path from u to a vertex w with $time(w) \geq time(v)$
  - A partial solution if it is on the only path leading to a vertex with $time(w) \geq time(v)$

Consequently, property maps may be modified based upon the time of the finished vertices and the current set of paths.

Graph Concepts for a Generic Viterbi Algorithm

Trellis graph

- Incidence Graph
- Vertex: Time and State
- Adjacent vertices
  - Dynamic: range of vertex states generated by factory classes
  - Time-dependent: factory generated by a factory
  - Vertex times generated separately: another factory concept

Optimal path algorithm

- Requires only an Incidence Graph

All an implementation detail: library composes graphs and algorithms from application-specific classes modeling concepts

Property Map Concepts for a Generic Viterbi Algorithm

Property maps

- Read-writable property maps
- Trimmable property maps
- An implementation detail

Edge weight

- Graph algorithms require a readable property map
- A Binary Function is a more natural expression
- Library encapsulates a Binary Function as a property map

viterbi

`viterbi` is an overloaded set of functions implementing the Viterbi algorithm.

Notation

`f,l`	Objects of type `viterbi::pair`, i.e., vertices
`r`	An object modeling the Range Factory Factory Concept
`t`	An object modeling the Time Factory Concept
`e`	An object modeling the Edge Weight Concept
`c`	An object modeling the Consumer Concept (optional)
`p`	An object modeling the Policy Concept (optional)

Valid expressions

Expression	Result
`viterbi(f,l,r,t,e,c,p)`	`e::result_type`

Time Concept

Types modeling the Time Concept compose part of each trellis graph vertex.

Concept requirements

- Default constructible
- Copy constructible (Standard ref 20.1.3)
- Assignable (Standard ref 23.1)
- Equality comparable (Standard ref 20.1.1)
- Less than comparable (Standard ref 20.1.2)

State Concept

Types modeling the State Concept compose part of each trellis graph vertex.

Concept requirements

- Default constructible
- Copy constructible (Standard ref 20.1.3)
- Assignable (Standard ref 23.1)
- Equality comparable (Standard ref 20.1.1)
- Less than comparable (Standard ref 20.1.2)

State Iterator Concept

Out edge iterators for a Trells Graph must iterate over the states of the adjacent vertices. The type modeling this concept must be used consistently throughout.

Concept requirements

- Forward iterator (Standard ref 24.1.3 Table 74)
- `std::iterator_traits<StateIterator>::value_type` must be the type used throughout to model the State Concept.

Edge Weight Function Concept

A type modeling the Edge Weight Function Concept defines the edge weight of each edge in a trellis graph. Semantically, this concept corresponds to a function taking the source and target vertices of an edge in a trellis graph and returning the weight of that edge. This weight will be accumulated with the score of the source vertex to obtain a potential score for the target vertex.

Concept requirements

- Default constructible
- Adaptable binary function
  - First argument type: `viterbi::pair< Time,State >::type`
  - Second argument type: `viterbi::pair< Time,State >::type`
  - Result type: unspecified

Time Factory Concept

Semantically, this concept corresponds to a function taking the states of the source and target vertices, and the time of the source vertex of an edge in a trellis graph and returning the time of the target vertex.

Concept requirements

- Default constructible
- Ternary function: an obvious extension of the Unary and Binary Function Concepts.
  - First argument type: `State`
  - Second argument type: `State`
  - Third argument type: `Time`
  - Result type: `Time`

State Range Factory Concept

A type modeling the State Range Factory Concept defines the set of states for the vertices adjacent to a trellis graph vertex. Semantically, this concept corresponds to a function taking the state of the source vertex in a trellis graph and returning a range of states covering all of the adjacent vertices.

Concept requirements

- Default constructible
- Adaptable unary function
  - Argument type: `State`
  - Result type: `viterbi::state_range< StateIterator >::type`

State Range Factory Factory Concept

A type modeling the State Range Factory Factory Concept defines the means of constructing adjacent vertices in a trellis. Semantically, this concept corresponds to a function taking the time of a trellis graph vertex and returning a type modeling the State Range Factory Concept.

Concept requirements

- Default constructible
- Adaptable unary function
  - Argument type: `Time`
  - Result type: model of the State Range Factory Concept

Pathway Consumer Concept

A type modeling the Pathway Consumer Concept is an optional argument to `viterbi::viterbi()`. It will be called once for each vertex in the optimal path, starting with the head of the path and progressing in order toward the tail. The function will be called whenever vertices are trimmed from the head of the optimal path.

Concept requirements

- Default constructible
- Binary function
  - First argument type: `Time`
  - Second argument type: `State`
  - Result type: void

Policy Concept

A type modeling the Policy Concept is an optional argument to `viterbi::viterbi()`. This concept bundles together the main components controling how the optimal path algorithm will be constructed. These components are more fully described by the documentation for the Dijkstra shortest paths algorithm in the Boost Graph Library.

Concept requirements

- Default constructible
- Copy constructible (Standard ref 20.1.3)
- `Monoid monoid() const`
- `Compare compare() const`
- `Value identity() const`
- `Value worst_value() const`

Generic Viterbi Applications

Given these concepts, how can they be used to compose an application?

Example applications

- Convolution decoder
- Sequence alignment / edit distance

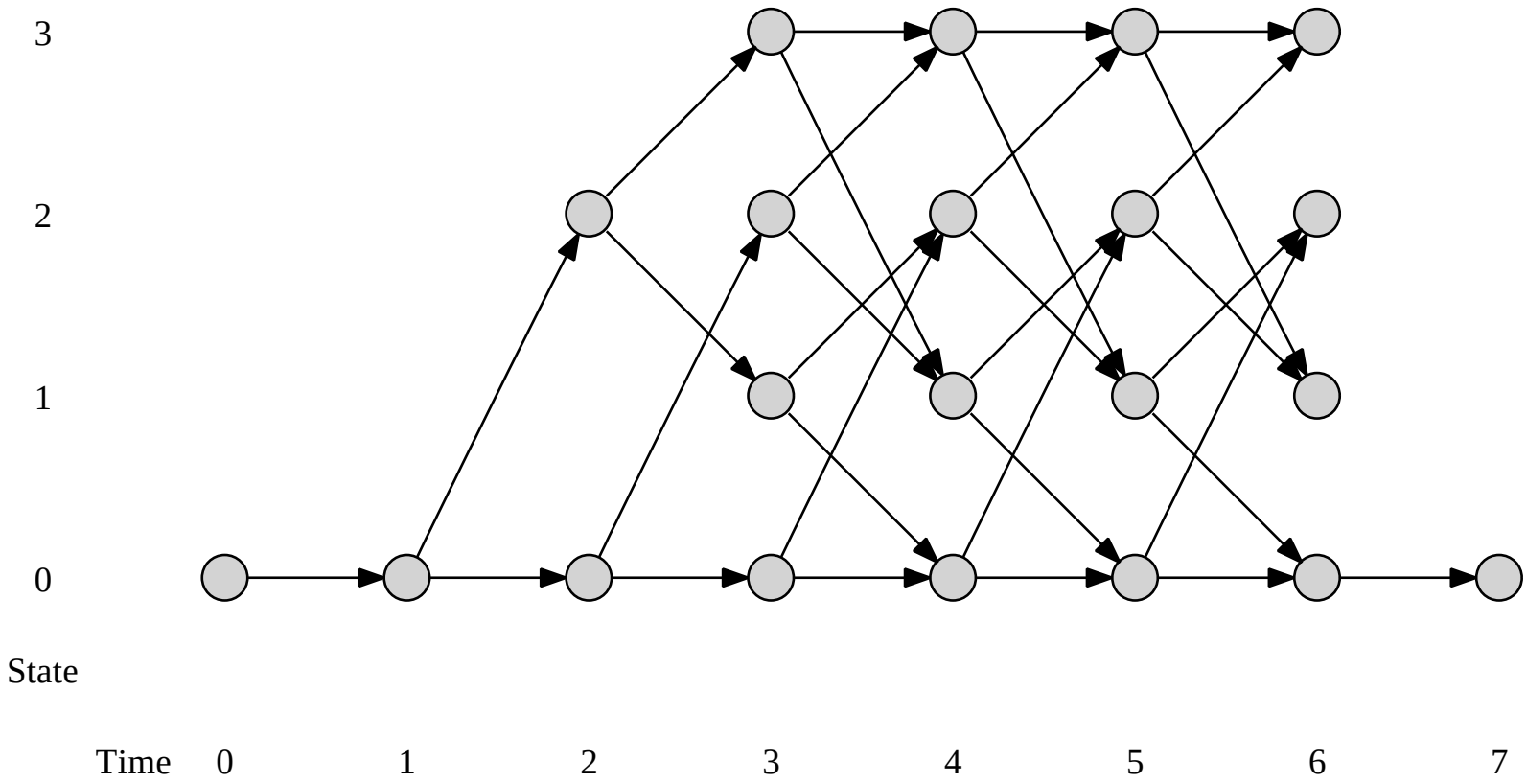
Goals

- Illustrate actual uses of the generic Viterbi code
- Clarify usage requirements
- Reveal potential design improvements

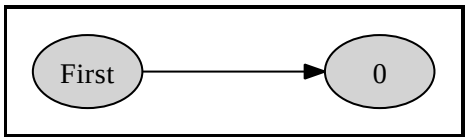
Convolution Decoder Example

- Problem: decode a sequence of encoded symbols
- Solution: optimal path through the appropriate trellis graph
- Challenge: how to describe the trellis graph
  - What is the shape of the graph?
  - How does its shape correspond to the graph-related concepts?
- Implementation: how are the required classes implemented?

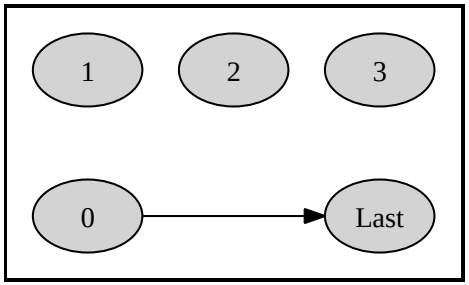
Trellis Graph: Convolution Decoder



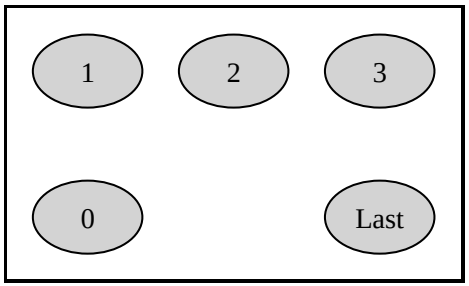
Finite State Machines: Decoder



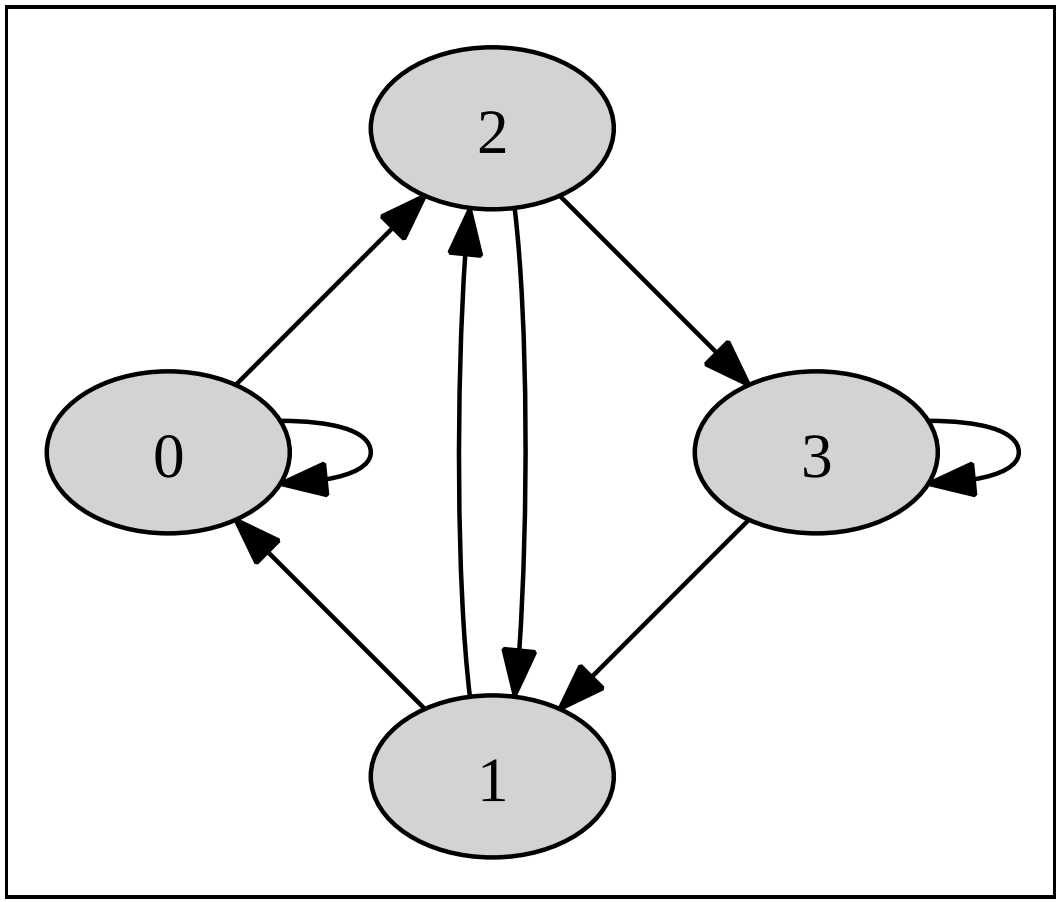
First time



End of sequence



Last time



Most times

Convolution Decoder Implementation

Concepts required

- Time: iterators into the sequence of symbols
- State: shift register state
- State iterator: for range of adjacent vertices
- State range factory: range = f(state)
- State range factory factory: range factory = f(time)
- Time factory: time = f(state,state,time)
- Edge weight function: score = f(vertex,vertex)

State Range Factory

```
// construct range of vertices adjacent for most vertices
template < typename StateIterator >
class most_factory
{
  typedef std::iterator_traits< StateIterator > T;
  typedef typename T::value_type state_type;
public:
  typename viterbi::state_range< StateIterator >::type
  operator () (state_type const& s) const
  {
    return result_type
      (StateIterator(s >> 1), StateIterator((s >> 1)
        + (1 << StateIterator::size())));
  }
};
```

State Range Factory Factory

```
// construct appropriate range factory object
template < typename Time, typename StateIterator >
struct state_range_factory_factory
{
  state_range_factory< StateIterator >
  operator () (Time const& t) const
  {
    if (first_time(t))
      return first_factory_;
    else if (last_time(t))
      return last_factory_;
    /* ... */
    else
      return most_factory_;
  }
```

Time Factory

```
// construct time for target vertex
template < typename Time, typename State >
struct incrementing_time_factory
{
  Time
  operator () (State const&, State const&, Time t) const
  { return ++t; }
};
```

Edge Weight Function

```
// score an edge
template < typename Time, typename State >
struct edge_weight_function
{
  typedef typename viterbi::pair< Time,State >::type pair;
  typedef pair first_argument_type;
  typedef pair second_argument_type;
  typedef int result_type;

  result_type
  operator () (pair const& u, pair const& v) const
  {
    return n_bits_different(*v.time(),v.state());
  }
};
```

Calling the Viterbi Function

```
// construct the data-dependent times and states
time_range t ( /* ... */ );
states s     ( /* ... */ );

// execute the Viterbi algorithm
int viterbi_result = viterbi::viterbi
    (viterbi::make_pair(t.first(),s.first()),
     viterbi::make_pair(t.last(),s.last()),
     make_state_range_factory_factory(t,s),
     make_incrementing_time_factory(t.first(),s.first()),
     make_edge_weight_function(t.first(),t.end(),
                               s.first()));
```

Generic Implementation of the Viterbi Decoder

Overview

- Implementation of a few well-defined concepts
- Clear association between concepts and elements in problem domain
- Complete freedom from the underlying algorithms and data structures

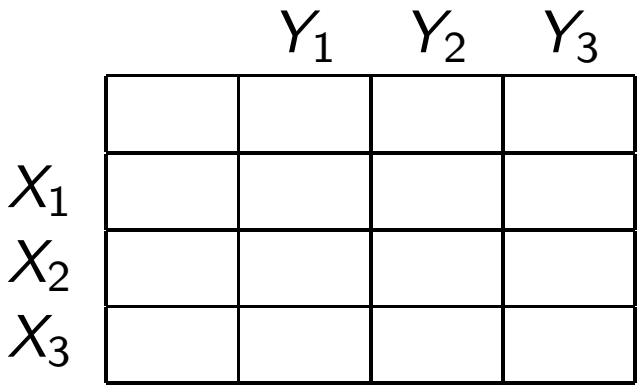
Sequence Alignment / Edit Distance Example

- Problem: align two sequences of symbols
- Traditional solution: matrix-based dynamic programming
- Challenge: how to describe the trellis graph
  - What is the shape of the graph?
  - How does its shape correspond to the graph-related concepts?
- Implementation: how to implement the required classes?

Illustrates applicability of concepts to a problem with a completely different structure that is normally solved in a completely different way

Are the concepts introduced actually generally applicable?

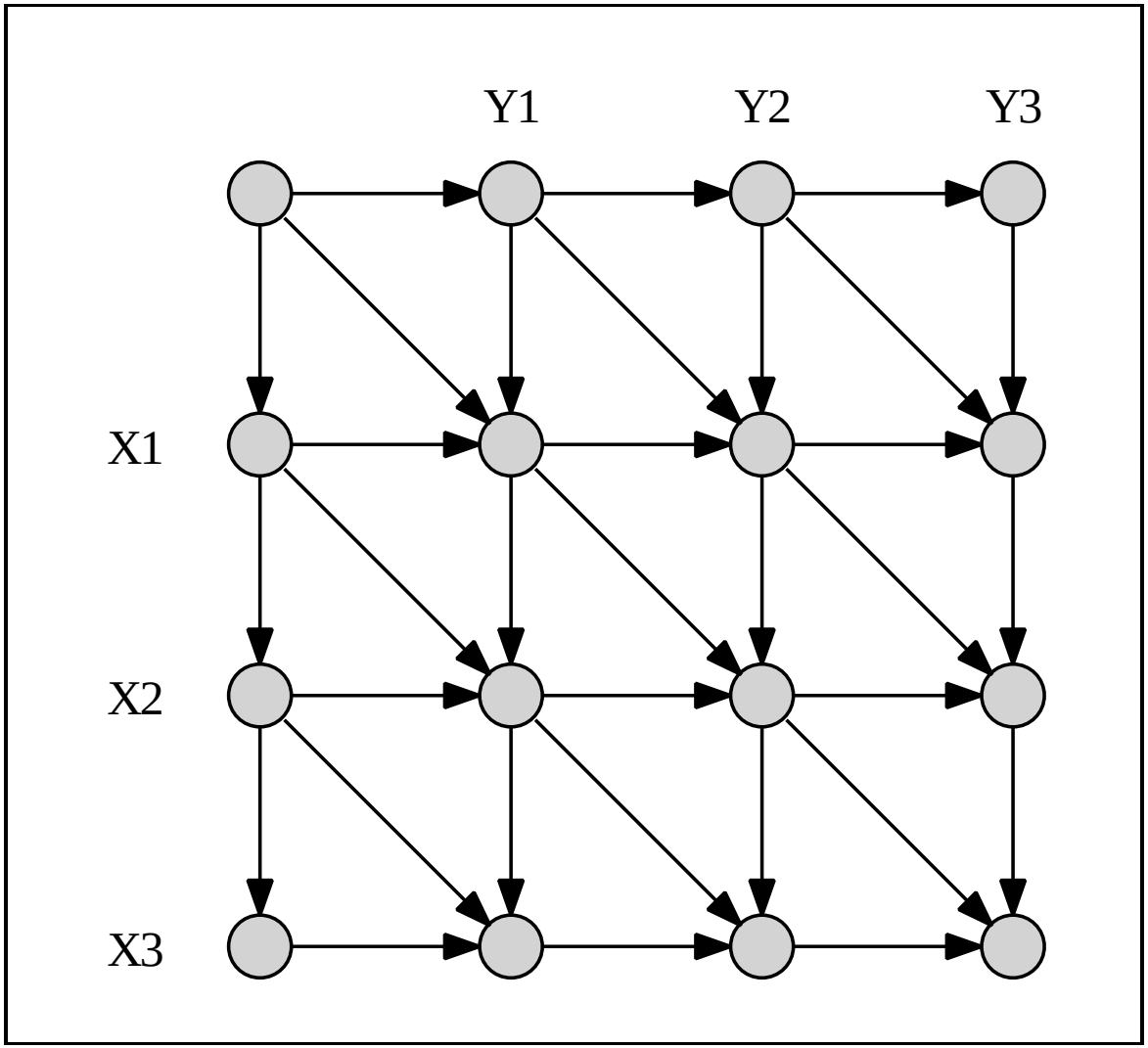
Dynamic Programming Matrix: Edit Distance



Traditional dynamic programming representation

- Alignment of the sequences $X_1X_2X_3$ and $Y_1Y_2Y_3$
- Matrix for scores of partial alignments
- Best alignment score: lower right corner element

Traditional Graph: Edit Distance

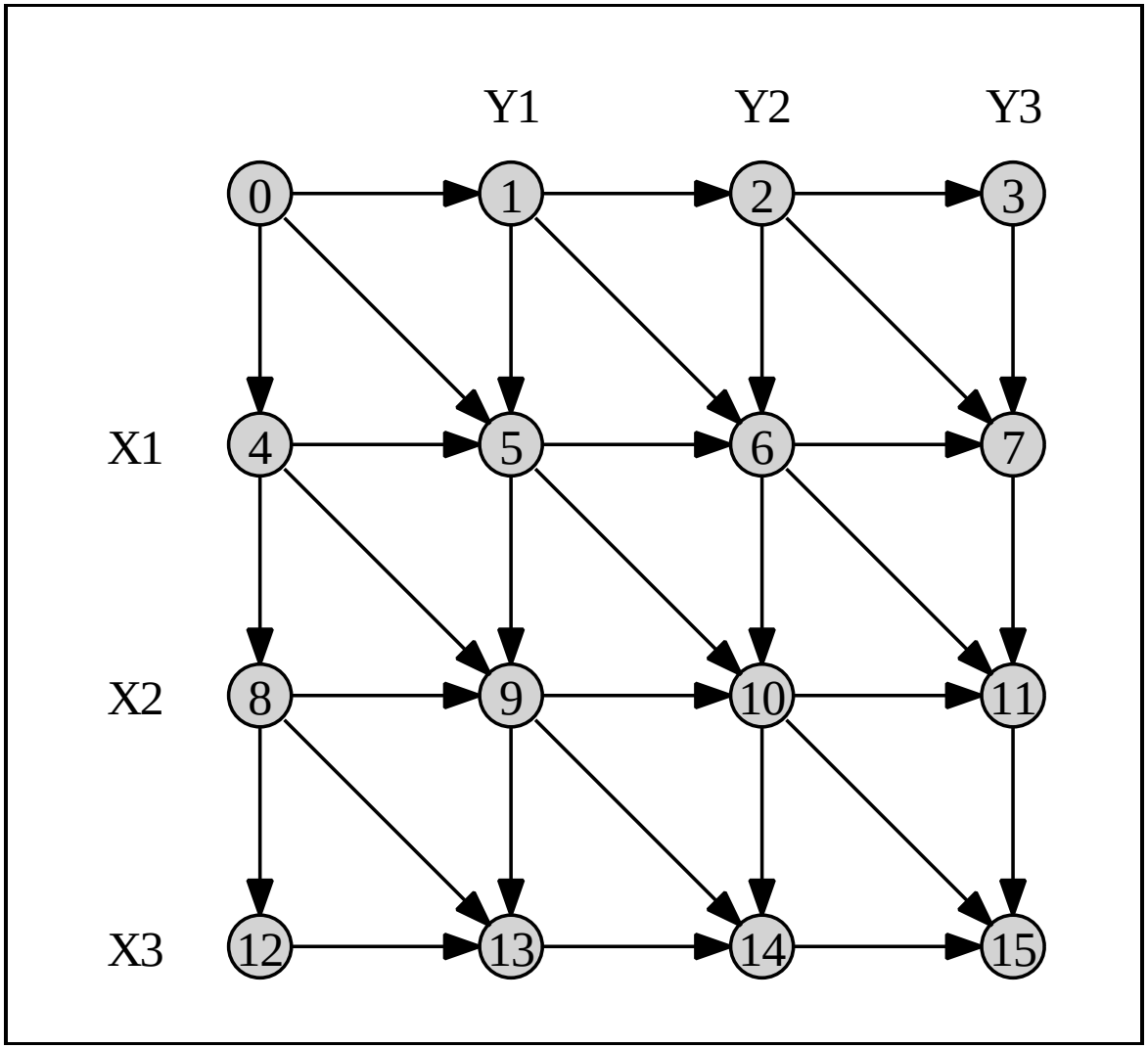


Traditional Graph: Edit Distance

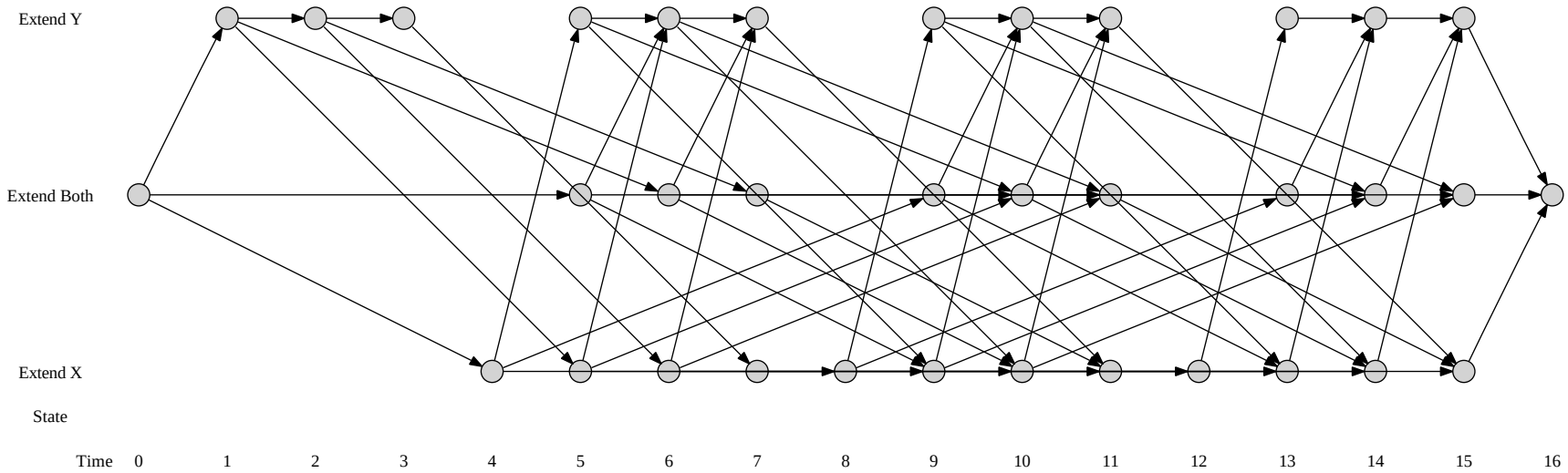
Characteristics

- Graph does not look like a trellis graph
  - The grpah is 2-dimensional like a trellis graph, but ...
  - What corresponds to the time axis?
  - What corresponds to the state axis?
- Graph vertices are insufficient to describe a subproblem
  - Each vertex corresponds to the alignment of two substrings
  - Each vertex defines the aligned substrings by specifying their ends
  - But, each vertex does not uniquely specify the problem that was solved
  - What does the vertex (X_2, Y_3) signify?

Traditional Graph: Edit Distance



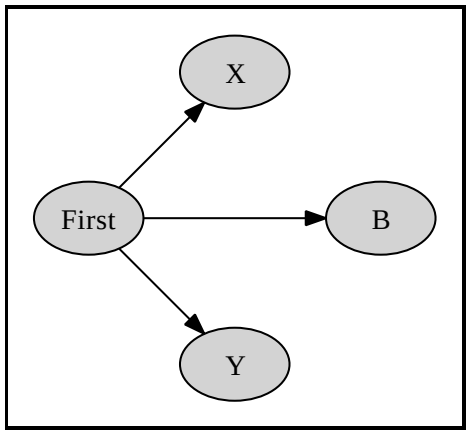
Trellis Graph: Edit Distance



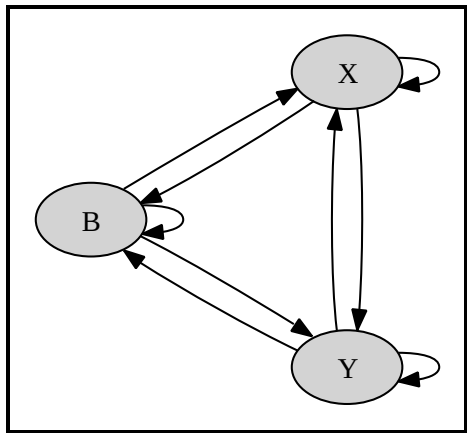
Complete trellis graph

- Alignment of the sequences $X_1X_2X_3$ and $Y_1Y_2Y_3$

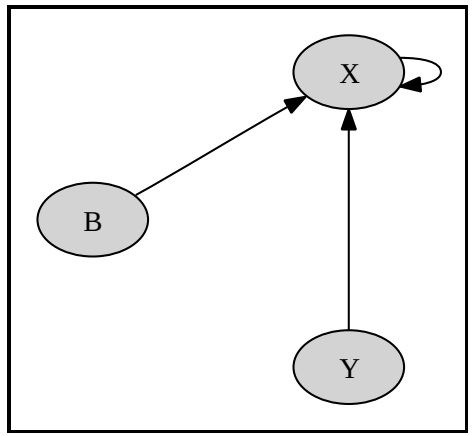
Finite State Machines: Edit Distance



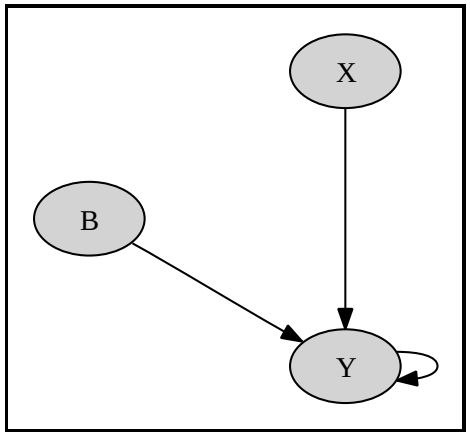
First time



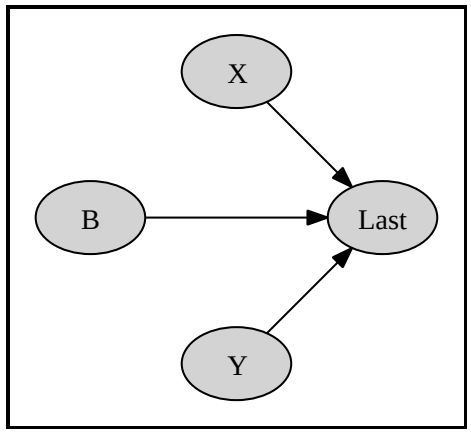
Most times



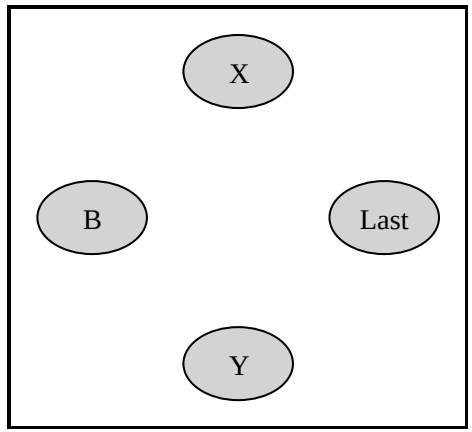
Right column



Bottom row



Corner vertex



Last time

Edit Distance Implementation

Concepts required

- Time — pair of iterators into the sequences to align
- State — first, x, y, both, last
- State range factory factory
- Time factory
- Edge weight function

State Range Factory Factory

```
// construct appropriate range factory object
template < typename Time, typename StateIterator >
struct state_range_factory_factory
{
  state_range_factory< StateIterator >
  operator () (Time const& t) const
  {
    return
        first_time(t) ?
          make_constant_range_factory(first_)
      : right_column_time(t) ?
          make_constant_range_factory(right_column_)
        /* ... */
      : make_associative_container_range_factory(most_);
  }
```

Time Factory

```
// construct time for target vertex
template < typename Time, typename State >
struct time_factory
{
  Time
  operator () (State const& u, State const& v,
               Time const& tu) const
  {
    return v.state() == State::B? tu.diagonal()
         : v.state() == State::X? tu.vertical()
         : v.state() == State::Y? tu.horizontal()
         :                        tu.last();
  }
};
```

Edge Weight Function

```
// score an edge
template < typename Time, typename State >
struct edge_weight_function
{
  typedef typename viterbi::pair< Time,State >::type pair;
  /* typedefs required for adaptable binary function */
  result_type
  operator () (pair const& u, pair const& v) const
  {
    return
      v.state() == State::B &&  v.match()? match_score_
    : v.state() == State::B && !v.match()? mismatch_score_
    : gap_score_;
  }
};
```

Calling the Viterbi Function

```
// construct the data-dependent times, states, weights
time_range t                        ( /* ... */ );
states s                            ( /* ... */ );
edge_weight_function lambda ( /* ... */ );

// execute the Viterbi algorithm
int viterbi_result = viterbi::viterbi
    (viterbi::make_pair(t.first(),s.first()),
     viterbi::make_pair(t.last(),s.last()),
     make_state_range_factory_factory(t,s),
     make_time_factory(t.first(),s.first()),
     lambda);
```

Generic Implementation of Sequence Alignment

Overview

- Implementation of a few well-defined concepts
- Clear association between concepts and elements in problem domain
- Complete freedom from the underlying algorithms and data structures

Generic Implementation of the Viterbi Algorithm

- Implementation of a few well-defined concepts
  - Time, state, state iterator, state factories, time factory, edge weights
- Clear association between concepts and elements in problem domain
- Almost complete freedom of concept implementation
  - Types used to implement concepts under user control
  - Library classes to ease integration between application-specific classes and library classes
- Complete distinction between application code and algorithm code
  - Removes algorithm concerns from application developers
  - Enables algorithm improvements independently of applications
  - Applications leverage algorithm advances transparently

Generic Viterbi Library: The Questions

- Is overall conceptualization correct and general?
- Are design decisions appropriate?
  - Nomenclature: namespaces and classes
  - Functionality: class organization
- What additional application examples are useful?
  - Pattern matching
  - Other trellis graph structures?

Generic Viterbi Library: The Future

- Is the Viterbi library appropriate for Boost?
- What should be improved prior to review?
- What points are likely to arise during review?
- Should elements be integrated into existing libraries?
  - Boost Graph Library
  - Property Map Library
  - Concept Checking Library

Generic Viterbi Library: The Potential

- Allows generic solutions to dynamic programming problems
- Minimum space utilization automatically guarranteed
- Enables rapid prototyping of new model structures
- Provides framework for exploring "model space" rather than implementation details
- Leverages algorithm expertise for domain-specific applications
- Encourage Boost to support scientific programming libraries