Design and Implementation of Speech Recognition Systems

Spring 2013

Class 27: Rescoring, Nbest and Confidence
29 Apr 2013
Topics

• The backpointer table as a directed acyclic graph
• N-best path search through a graph
  – Stack decoder
  – A*
• Confidence estimation
  – Forward Backward algorithm
• Acoustic Rescoring
The Directed Acyclic Graph

- A graph where every edge has a direction
- There are *no loops*
  - There is no path that revisits a node
- Such a graph *must* have some nodes that are purely *source* nodes
  - No incoming edges
- It also *must* have some nodes that are purely *sink* nodes
  - No outgoing edges
The **Directed Acyclic Graph**

- A graph where every edge has a direction
- Nodes may have *node cost*
- Edges may have *edge cost*
- Strictly equivalent: a graph with only edge costs
  - Node costs “pushed” onto edges
The *shortest-path* problem

- What is the shortest path from one node to another?
  - “Shortest” $\Rightarrow$ least-cost
  - Could also mean *most-score*
    - If values assigned to nodes and edges represent scores instead of costs
    - We’ll assume costs for now; easily modified to deal with scores
      - Simply flip “min” to “max” and vice versa
Dijkstra’s algorithm (1959)

- Gives the cost of the shortest path between two nodes
- Condition: All costs are positive
  - I.e. traversing an edge is expensive
  - Addendum: visiting a node is expensive
    - For a DAG, node costs can be converted to edge costs by simply adding them to all outgoing edges from that node
  - Passing through a node or edge can never be profitable
    - Can never have negative cost
**Dijkstra’s algorithm**

1. **Set current node to “visited” state**
   - Indicated by blue color
   - Cost of best path to current node is simply node cost
   - Set best path cost of *all other nodes* to infinity
Dijkstra’s algorithm

1. Set current node to “visited” state

2. **Extend paths from current node to all of its unvisited children**
   - Add edge cost + node cost to get “current” path cost
   - If current path cost to a node is *lower* than existing path cost, replace existing path cost with current path cost
1. Set current node to “visited” state
2. Extend paths from current node to all of its unvisited children
3. Select the “unvisited” node with lowest cost: set it to “visited”
   - If this is the destination node, terminate; shortest path cost found
   - If the lowest cost unvisited node has a cost of infinity, fail (no path).
1. Set current node to “visited” state
2. Extend paths from current node to all of its unvisited children
3. Select the “unvisited” node with lowest cost: set it to “visited”
   – If this is the destination node, terminate; shortest path cost found
   – If the lowest cost unvisited node has a cost of infinity, fail (no path)
4. Set the node to “current” and return to 2
Dijkstra’s algorithm

1. Set current node to “visited” state
2. Extend paths from current node to all of its unvisited children
3. **Select the “unvisited” node with lowest cost: set it to “visited”**
   - If this is the destination node, terminate; shortest path cost found
   - If the lowest cost unvisited node has a cost of infinity, fail (no path)
4. Set the node to “current” and return to 2
Dijkstra’s algorithm

1. Set current node to “visited” state
2. Extend paths from current node to all of its unvisited children
3. Select the “unvisited” node with lowest cost: set it to “visited”
   - If this is the destination node, terminate; shortest path cost found
   - If the lowest cost unvisited node has a cost of infinity, fail (no path)
4. Set the node to “current” and return to 2
Dijkstra's algorithm

1. Set current node to “visited” state
2. Extend paths from current node to all of its unvisited children
3. Select the “unvisited” node with lowest cost: set it to “visited”
   - If this is the destination node, terminate; shortest path cost found
   - If the lowest cost unvisited node has a cost of infinity, fail (no path)
4. Set the node to “current” and return to 2
Dijkstra’s algorithm

1. Set current node to “visited” state
2. Extend paths from current node to all of its unvisited children
3. Select the “unvisited” node with lowest cost: set it to “visited”
   - If this is the destination node, terminate; shortest path cost found
   - If the lowest cost unvisited node has a cost of infinity, fail (no path)
4. Set the node to “current” and return to 2
Dijkstra’s algorithm

Lowest Path

\[ \text{cost} = n_3 + e_6 + n_4 + e_7 + n_5 \]

1. Set current node to “visited” state
2. Extend paths from current node to all of its unvisited children
3. Select the “unvisited” node with lowest cost: set it to “visited”
   - If this is the destination node, terminate; shortest path cost found
   - If the lowest cost unvisited node has a cost of infinity, fail (no path)
4. Set the node to “current” and return to 2
Dijkstra’s algorithm

1. Set current node to “visited” state
2. Extend paths from current node to all of its unvisited children
3. Select the “unvisited” node with lowest cost: set it to “visited”
   - If this is the destination node, terminate; shortest path cost found
   - If the lowest cost unvisited node has a cost of infinity, fail (no path)
4. Set the node to “current” and return to 2
**Dijkstra’s Algorithm**

- Dijkstra’s algorithm can be continued to find the shortest path score from the source node to *all* nodes in the graph.

- Simply continue algorithm until either
  - All nodes are visited OR
  - All unvisited nodes have infinite cost

- **Computational Cost**
  - Naïve implementation $|V|^2$
    - $|V| = \text{no. of nodes}$
  - Optimal implementation: $|E| + |V| \log |V|$
    - $|E| = \text{no. of edges}$
Problem 2: Computing *Total* path cost

- What is the *total* cost of all paths from all source nodes to any particular node?
Problem 2: The forward algorithm

- Initialize: Set “total path score” for all nodes to 0
Problem 2: The forward algorithm

1. Mark source nodes
   – Source nodes have node scores
Problem 2: The forward algorithm

1. Mark source nodes
2. Extend paths from all source nodes to all children nodes
   - Update node scores
Extending a path: Cost of extended path:
- Path score = $f_{\text{ext}}(\text{current path score}, \text{edge score}, \text{node score})$
- Typically $f_{\text{ext}}(a,b,c) = a+b+c$ or $a*b*c$
- If edge and node scores are probabilities, we use $a*b*c$

Converging paths: If K paths converge on a node, node score is:
- node score = node score + $f_{\text{node}}(\text{path score1}, \text{path score2})$
- If node and edge scores are probabilities, we use $f_{\text{node}}(a,b,c) = a+b+c$
Problem 2: The forward algorithm

1. Mark source nodes

2. Extend paths from all source nodes to all children nodes

3. Mark utilized sources and edges as “evaluated”
   - Mark all utilized edges as “evaluated”
   - Mark all current source nodes as “evaluated”
1. Mark source nodes
2. Extend paths from all source nodes to all children nodes
3. Mark utilized sources and edges as “evaluated”
4. Mark all children nodes such that all incoming edges are evaluated as “source” nodes
Problem 2: The forward algorithm

1. Mark source nodes
2. Extend paths from all source nodes to all children nodes
3. Mark utilized sources and edges as “evaluated”
4. Mark all children nodes such that all incoming edges are evaluated as “source” nodes
5. **If any “unevaluated” source nodes remain, return to 2, otherwise terminate**
Problem 2: The forward algorithm

1. Mark source nodes
2. Extend paths from all source nodes to all children nodes
3. Mark utilized sources and edges as “evaluated”
4. Mark all children nodes such that all incoming edges are evaluated as “source” nodes
5. If any “unevaluated” source nodes remain, return to 2, otherwise terminate
1. Mark source nodes
2. Extend paths from all source nodes to all children nodes
3. Mark utilized sources and edges as “evaluated”
4. Mark all children nodes such that all incoming edges are evaluated as “source” nodes
5. If any “unevaluated” source nodes remain, return to 2, otherwise terminate
Problem 2: The forward algorithm

1. Mark source nodes
2. Extend paths from all source nodes to all children nodes
3. Mark utilized sources and edges as “evaluated”
4. Mark all children nodes such that all incoming edges are evaluated as “source” nodes
5. If any “unevaluated” source nodes remain, return to 2, otherwise terminate
Problem 2: The forward algorithm

1. Mark source nodes
2. Extend paths from all source nodes to all children nodes
3. Mark utilized sources and edges as “evaluated”
4. Mark all children nodes such that all incoming edges are evaluated as “source” nodes
5. If any “unevaluated” source nodes remain, return to 2, otherwise terminate
Problem 2: The forward algorithm

1. Mark source nodes
2. Extend paths from all source nodes to all children nodes
3. Mark utilized sources and edges as “evaluated”
4. Mark all children nodes such that all incoming edges are evaluated as “source” nodes
5. If any “unevaluated” source nodes remain, return to 2, otherwise terminate
Problem 2: The forward algorithm

1. Mark source nodes
2. Extend paths from all source nodes to all children nodes
3. Mark utilized sources and edges as “evaluated”
4. Mark all children nodes such that all incoming edges are evaluated as “source” nodes
5. If any “unevaluated” source nodes remain, return to 2, otherwise terminate
Problem 2: The forward algorithm

1. Mark source nodes
2. Extend paths from all source nodes to all children nodes
3. Mark utilized sources and edges as “evaluated”
4. Mark all children nodes such that all incoming edges are evaluated as “source” nodes
5. If any “unevaluated” source nodes remain, return to 2, otherwise terminate
Problem 2: The forward algorithm

1. Mark source nodes
2. Extend paths from all source nodes to all children nodes
3. Mark utilized sources and edges as “evaluated”
4. Mark all children nodes such that all incoming edges are evaluated as “source” nodes
5. If any “unevaluated” source nodes remain, return to 2, otherwise terminate
Problem 2: The forward algorithm

1. Mark source nodes
2. Extend paths from all source nodes to all children nodes
3. Mark utilized sources and edges as “evaluated”
4. Mark all children nodes such that all incoming edges are evaluated as “source” nodes
5. **If any “unevaluated” source nodes remain, return to 2, otherwise terminate**
Problem 2: The forward algorithm

1. Mark source nodes
2. Extend paths from all source nodes to all children nodes
3. Mark utilized sources and edges as “evaluated”
4. Mark all children nodes such that all incoming edges are evaluated as “source” nodes
5. If any “unevaluated” source nodes remain, return to 2, otherwise terminate
Problem 2: The forward algorithm

1. Mark source nodes
2. Extend paths from all source nodes to all children nodes
3. Mark utilized sources and edges as “evaluated”
4. Mark all children nodes such that all incoming edges are evaluated as “source” nodes
5. **If any “unevaluated” source nodes remain, return to 2, otherwise terminate**
Problem 2: The forward algorithm

1. Mark source nodes
2. Extend paths from all source nodes to all children nodes
3. Mark utilized sources and edges as “evaluated”
4. Mark all children nodes such that all incoming edges are evaluated as “source” nodes
5. If any “unevaluated” source nodes remain, return to 2, otherwise terminate
Problem 2: The forward algorithm

1. Mark source nodes
2. Extend paths from all source nodes to all children nodes
3. Mark utilized sources and edges as “evaluated”
4. Mark all children nodes such that all incoming edges are evaluated as “source” nodes
5. If any “unevaluated” source nodes remain, return to 2, otherwise terminate
At termination: The final score of any node is the total path score of all paths from all source nodes to that node.

The total score of all sink nodes is the total score of all paths through the graph.
The Backward Scores

• The forward algorithm computes total score of all paths from sources to any node

• We can similarly compute the total score of all paths from a node to all sink nodes

• This is computed using a backward algorithm
Problem 3: The backward algorithm

- Initialize: Set “total path score” for all nodes to 0
1. Mark sink nodes

- Sink nodes have node scores

$B_7 = n_7$
Problem 3: The backward algorithm

1. Mark sink nodes

2. Extend paths *backwards* from all sink nodes to all *parent* nodes
   - Update node scores similarly to the forward algorithm
Problem 3: The backward algorithm

- Extending a path: Cost of extended path:
  - Path score = $f_{\text{ext}}(\text{current path score, edge score, node score})$
  - Typically $f_{\text{ext}}(a,b,c) = a+b+c$ or $a*b*c$
  - If edge and node scores are probabilities, we use $a*b*c$

- Converging paths: If $K$ paths converge on a node, node score is:
  - node score = node score + $f_{\text{node}}(\text{path score1, path score2})$
  - For probabilistic graphs, $f_{\text{node}}(a,b,c) = a+b+c$
Problem 3: The backward algorithm

1. Mark sink nodes
2. Extend paths *backwards* from all sink nodes to all *parent* nodes
3. **Mark utilized sources and edges as “evaluated”**
   - Mark all utilized edges as “evaluated”
   - Mark all current source nodes as “evaluated”
1. Mark sink nodes
2. Extend paths *backwards* from all sink nodes to all *parent* nodes
3. Mark utilized sources and edges as “evaluated”
4. **Mark all *parent* nodes such that all outgoing edges are evaluated as “sink” nodes**
Problem 3: The backward algorithm

1. Mark sink nodes
2. Extend paths *backwards* from all sink nodes to all *parent* nodes
3. Mark utilized sources and edges as “evaluated”
4. Mark all *parent* nodes such that all outgoing edges are evaluated as “sink” nodes
5. If any “unevaluated” source nodes remain, return to 2, otherwise terminate
1. Mark sink nodes
2. Extend paths *backwards* from all sink nodes to all *parent* nodes
3. Mark utilized sources and edges as “evaluated”
4. Mark all *parent* nodes such that all outgoing edges are evaluated as “sink” nodes
5. If any “unevaluated” source nodes remain, return to 2, otherwise terminate
Problem 3: The backward algorithm

1. Mark sink nodes
2. Extend paths backwards from all sink nodes to all parent nodes
3. Mark utilized sources and edges as “evaluated”
4. Mark all parent nodes such that all outgoing edges are evaluated as “sink” nodes
5. If any “unevaluated” source nodes remain, return to 2, otherwise terminate
Problem 3: The backward algorithm

1. Mark sink nodes
2. Extend paths *backwards* from all sink nodes to all *parent* nodes
3. Mark utilized sources and edges as “evaluated”
4. Mark all *parent* nodes such that all outgoing edges are evaluated as “sink” nodes
5. If any “unevaluated” source nodes remain, return to 2, otherwise terminate
The Backward Algorithm

• At termination: The final score of any node is the total path score of all paths from that node to all sink nodes
The Forward-Backward Scores

• We can now compute the total score of all paths from all sources to all sinks that pass through a specific node
• Total cost of all paths through node 4 = \( S_4 \times B_4 / n_4 \)
  - In general, for any node \( i \), total cost = \( S_i \times B_i / n_i \)
  - Assuming probability-based combination

• Forward score * Backward score / node score
  - \( S_i \) = forward score, \( B_i \) = backward score; \( n_i \) = node score
  - Must divide out \( n_i \) since it is included in both forward and backward scores
  - Division eliminates duplication
YET another graph problem: N-best

• We have seen how to find the score the shortest path between a source and a sink node
  – And, consequently, the shortest path itself

• But what is the length of the *second* shortest path?
  – Or the N-th shortest path
  – *What are these paths?*
The \textit{n-shortest paths} problem

- What are the \( N \) shortest paths between the source and sink nodes?
  - The “Stack” decoder
  - The “A*” algorithm
The stack decoder

- Begin at the source
  - The total cost of the path thus far is simply $n_1$
    - $S_1 = n_1$
  - Push “1:S$_1$” into a “stack”
    - “1” identifies the path, $S_1$ is its score
The stack decoder

1. Pop current shortest partial path from stack
The stack decoder

1. Pop current shortest partial path from stack
2. If: final node of partial path is sink node, output it
   - If desired number (N) of outputs obtained: terminate
   - else: return to 1.
3. else: go to 3
1. Pop current shortest partial path from stack
2. If: final node of partial path is sink node, output it
   – If desired number (N) of outputs obtained: terminate
   else: return to 1.
else: go to 3
3. Extend partial path by expanding all edges of final node on path
The stack decoder

1. Pop current shortest partial path from stack
2. If : final node of partial path is sink node, output it
   - If desired number (N) of outputs obtained : terminate
   else: return to 1.
   else: go to 3
3. Extend partial path by expanding all edges of final node on path
4. Push all extended paths into stack
   - Arrange stack by increasing cost: lowest cost path on top
1. Pop current shortest partial path from stack
2. If: final node of partial path is sink node, output it
   – If desired number (N) of outputs obtained: terminate
   else: return to 1.
else: go to 3
3. Extend partial path by expanding all edges of final node on path
4. Push all extended paths into stack
   – Arrange stack by increasing cost: lowest cost path on top
5. Return to 1.
1. Pop current shortest partial path from stack
2. If: final node of partial path is sink node, output it
   – If desired number (N) of outputs obtained: terminate
   else: return to 1.
else: go to 3
3. Extend partial path by expanding all edges of final node on path
4. Push all extended paths into stack
   – Arrange stack by increasing cost: lowest cost path on top
5. Return to 1.
The stack decoder

1. Pop current shortest partial path from stack
2. If: final node of partial path is sink node, output it
   – If desired number (N) of outputs obtained: terminate
   else: return to 1.
   else: go to 3
3. Extend partial path by expanding all edges of final node on path
4. Push all extended paths into stack
   – Arrange stack by increasing cost: lowest cost path on top
5. Return to 1.

\[ S_{1,3} = S_{1,3} + e_6 + n_4 \]
\[ S_{1,3,6} = S_{1,3} + e_4 + n_6 \]
The *stack decoder*

1. Pop current shortest partial path from stack
2. If: final node of partial path is sink node, output it
   - If desired number (N) of outputs obtained: terminate
   - else: return to 1.
   - else: go to 3
3. Extend partial path by expanding all edges of final node on path
4. **Push all extended paths into stack**
   - Arrange stack by increasing cost: lowest cost path on top
5. Return to 1.
1. Pop current shortest partial path from stack
2. If: final node of partial path is sink node, output it
   – If desired number (N) of outputs obtained: terminate
   else: return to 1.
else: go to 3
3. Extend partial path by expanding all edges of final node on path
4. Push all extended paths into stack
   – Arrange stack by increasing cost: lowest cost path on top
5. Return to 1.
The stack decoder

1. Pop current shortest partial path from stack
2. If: final node of partial path is sink node, output it
   – If desired number (N) of outputs obtained: terminate
   else: return to 1.
else: go to 3
3. Extend partial path by expanding all edges of final node on path
4. Push all extended paths into stack
   – Arrange stack by increasing cost: lowest cost path on top
5. Return to 1.
The stack decoder

1. Pop current shortest partial path from stack
2. If: final node of partial path is sink node, output it
   – If desired number (N) of outputs obtained: terminate
   else: return to 1.
else: go to 3
3. Extend partial path by expanding all edges of final node on path
4. Push all extended paths into stack
   – Arrange stack by increasing cost: lowest cost path on top
5. Return to 1.

1, 2: $S_{1,2}$
1, 3, 6: $S_{1,3,6}$

$1,3,4: S_{1,3,4}$

$S_{1,3,4,5} = S_{1,3,4} + e_7 + n_5$
$S_{1,3,4,6} = S_{1,3,4} + e_8 + n_6$
$S_{1,3,4,7} = S_{1,3,4} + e_9 + n_7$
1. Pop current shortest partial path from stack
2. If : final node of partial path is sink node, output it
   – If desired number (N) of outputs obtained : terminate
   else: return to 1.
else: go to 3
3. Extend partial path by expanding all edges of final node on path
4. **Push all extended paths into stack**
   – Arrange stack by increasing cost: lowest cost path on top
5. Return to 1.
The stack decoder

1. Pop current shortest partial path from stack
2. If: final node of partial path is sink node, output it
   – If desired number (N) of outputs obtained: terminate
   else: return to 1.
else: go to 3
3. Extend partial path by expanding all edges of final node on path
4. Push all extended paths into stack
   – Arrange stack by increasing cost: lowest cost path on top
5. Return to 1.
The stack decoder

1. Pop current shortest partial path from stack
2. If: final node of partial path is sink node, output it
   – If desired number (N) of outputs obtained: terminate
   else: return to 1.
else: go to 3

3. Extend partial path by expanding all edges of final node on path
4. Push all extended paths into stack
   – Arrange stack by increasing cost: lowest cost path on top
5. Return to 1.
1. Pop current shortest partial path from stack
2. If: final node of partial path is sink node, output it
   -- If desired number (N) of outputs obtained: terminate
   else: return to 1.
else: go to 3
3. **Extend partial path by expanding all edges of final node on path**
4. Push all extended paths into stack
   -- Arrange stack by increasing cost: lowest cost path on top
5. Return to 1.
The stack decoder

1. Pop current shortest partial path from stack
2. If: final node of partial path is sink node, output it
   -- If desired number (N) of outputs obtained: terminate
   else: return to 1.
else: go to 3
3. Extend partial path by expanding all edges of final node on path
4. **Push all extended paths into stack**
   -- Arrange stack by increasing cost: lowest cost path on top
5. Return to 1.
The stack decoder

1. Pop current shortest partial path from stack
2. If : final node of partial path is sink node, output it
   – If desired number (N) of outputs obtained : terminate
   else: return to 1.
else: go to 3
3. Extend partial path by expanding all edges of final node on path
4. Push all extended paths into stack
   – Arrange stack by increasing cost: lowest cost path on top
5. Return to 1.
The stack decoder

1. Pop current shortest partial path from stack

2. **If**: final node of partial path is sink node, output it
   - If desired number (N) of outputs obtained: terminate
   - else: return to 1.

else: go to 3

3. Extend partial path by expanding all edges of final node on path

4. Push all extended paths into stack
   - Arrange stack by increasing cost: lowest cost path on top

5. Return to 1.
The stack decoder

1. Pop current shortest partial path from stack

2. If: final node of partial path is sink node, output it
   - If desired number (N) of outputs obtained: terminate
   - else: return to 1.

   else: go to 3

3. Extend partial path by expanding all edges of final node on path

4. Push all extended paths into stack
   - Arrange stack by increasing cost: lowest cost path on top

5. Return to 1.
1. Pop current shortest partial path from stack
2. If: final node of partial path is sink node, output it
   – If desired number \((N)\) of outputs obtained: terminate
   else: return to 1.
else: go to 3
3. Extend partial path by expanding all edges of final node on path
4. Push all extended paths into stack
   – Arrange stack by increasing cost: lowest cost path on top
5. Return to 1.
The stack decoder

1. Pop current shortest partial path from stack
2. If: final node of partial path is sink node, output it
   – If desired number (N) of outputs obtained: terminate
   else: return to 1.
else: go to 3

3. Extend partial path by expanding all edges of final node on path
4. Push all extended paths into stack
   – Arrange stack by increasing cost: lowest cost path on top
5. Return to 1.
The stack decoder

1. Pop current shortest partial path from stack
2. If: final node of partial path is sink node, output it
   — If desired number (N) of outputs obtained: terminate
   else: return to 1.
else: go to 3

3. **Extend partial path by expanding all edges of final node on path**
4. Push all extended paths into stack
   — Arrange stack by increasing cost: lowest cost path on top
5. Return to 1.

---

1,2,5: \( S_{1,2,5} \)
1,3,6: \( S_{1,3,6} \)
1,3,4,5: \( S_{1,3,4,5} \)
1,3,4,6: \( S_{1,3,4,6} \)

1,2,4: \( S_{1,2,4} \)
1,2,4,5: \( S_{1,2,4,5} \)
1,2,4,6: \( S_{1,2,4,6} \)
1,2,4,7: \( S_{1,2,4,7} \)
1,3,4,7: \( S_{1,3,4,7} \)
The stack decoder

- The algorithm continues until the desired number (N) of paths are output

- The process *guarantees* that these are the N shortest (lowest-cost) paths through the graph

- Computational complexity: Upper bound = N!
  - In practice, much much smaller

- Still, it has a problem:
  - Frequently its very slow
  - Why?...
Problem with the stack decoder

- The top of the stack is often dominated by paths that are close to the source
  - They tend to have lower costs than paths that are closer to the sink
- The algorithm will preferentially pop these
  - Effectively spending most of the time expanding shallow paths, instead of exploring the more promising deeper ones
The A* Algorithm

• Solution: *Predict the future*
  – Replace every score $S^*_{*,b}$ with $S^*_{*,b} + B_{b,sink}$
  – $S^*_{*,b}$ is the score of a path ending at node $b$
  – $B_{b,sink}$ is a *guess* of the lowest cost score from $b$ to the sink

• Note that setting $B_{b,sink} = 0$ results in the conventional stack decoder

• **Guarantee:** If $B_{b,sink}$ is a *true lower bound* on the best path score from $b$ to the sink, the A* algorithm returns the correct results
  – I.e. the same result as the stack decoder
  – Only much much faster
The $A^*$ algorithm

- First: Compute the best path cost from each node to the sink node
  - Can be computed using Dijkstra’s algorithm
• Begin at the source
  – The total cost of the path is the forward path cost to the node PLUS the (guessed) best path cost to the sink
  
  • \( S_1 = n_1 + B_1 \)
  – Push “1:S1” into a “stack”
  
  • “1” identifies the path, \( S_1 \) is its score
1. Pop current shortest partial path from stack

2. If: final node of partial path is sink node, output it
   – If desired number (N) of outputs obtained: terminate
     else: return to 1.
   else: go to 3

3. Extend partial path by expanding all edges of final node on path
• NOTE: Modified Score Computation
• Subtract $B_1$
  – Subtract \textit{previous} (best guess of) lowest cost of remaining path to sink
• Add $B_{\text{node}}$
  – Add \textit{current} (best guess of) of lowest cost of remaining path from current node to sink

$S_{1,2} = S_1 + e_1 + n_2 - B_1 + B_2$
$S_{1,3} = S_1 + e_2 + n_3 - B_1 + B_3$
1. Pop current shortest partial path from stack
2. If: final node of partial path is sink node, output it
   - If desired number (N) of outputs obtained: terminate
   else: return to 1.
else: go to 3
3. Extend partial path by expanding all edges of final node on path
4. Push all extended paths into stack
   - Arrange stack by increasing cost: lowest cost path on top
1. **Pop current shortest partial path from stack**

2. If: final node of partial path is sink node, output it
   - If desired number (N) of outputs obtained: terminate
   - else: return to 1.
   - else: go to 3

3. Extend partial path by expanding all edges of final node on path

4. Push all extended paths into stack
   - Arrange stack by increasing cost: lowest cost path on top

5. **Return to 1.**
1. Pop current shortest partial path from stack
2. If: final node of partial path is sink node, output it
   – If desired number (N) of outputs obtained: terminate
   else: return to 1.
else: go to 3
3. Extend partial path by expanding all edges of final node on path
4. Push all extended paths into stack
   – Arrange stack by increasing cost: lowest cost path on top
5. Return to 1.
1. Pop current shortest partial path from stack
2. If: final node of partial path is sink node, output it
   – If desired number (N) of outputs obtained: terminate
   else: return to 1.
else: go to 3
3. **Extend partial path by expanding all edges of final node on path**
4. Push all extended paths into stack
   – Arrange stack by increasing cost: lowest cost path on top
5. Return to 1.
The A* Algorithm

- The A* proceeds as the stack decoder does, with the modification that predicted future scores are always incorporated.

- Caveats:
  - For the predicted future score do *not* include the score of the first node.
    - To ensure that the node score is not included twice in any path score.
    - E.g., $B_3$ must not include $n_3$.
    - This can be done by explicitly subtracting out $n_3$ from the best path score computed by Dijkstra’s algorithm.
Returning to ASR

• We now apply what we have learned to address some problems in speech recognition
  
• N-best generation
• Rescoring
• Confidence estimation
The Backpointer Table is a Tree

<table>
<thead>
<tr>
<th>Id</th>
<th>Time</th>
<th>Parent</th>
<th>Score</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>&lt;s&gt;</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>s1</td>
<td>SIL</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>s2</td>
<td>APRKT</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>s3</td>
<td>APL</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>s4</td>
<td>SIL</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>s5</td>
<td>APRKT</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>0</td>
<td>s6</td>
<td>APL</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>6</td>
<td>s7</td>
<td>SIL</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
<td>s8</td>
<td>APRKT</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>4</td>
<td>s9</td>
<td>APL</td>
</tr>
</tbody>
</table>

Note LM probabilities now
The BackPointer Table is a TREE

• The backpointer table is a tree
The BackPointer Table is a TREE

• The backpointer table is a tree

<table>
<thead>
<tr>
<th>Id</th>
<th>time</th>
<th>parent</th>
<th>score</th>
<th>word</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>&lt;s&gt;</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>s1</td>
<td>SIL</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>s2</td>
<td>APRICOT</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>s3</td>
<td>APPLE</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>s4</td>
<td>SIL</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>s5</td>
<td>APRICOT</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>0</td>
<td>s6</td>
<td>APPLE</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>6</td>
<td>s7</td>
<td>SIL</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
<td>s8</td>
<td>APRICOT</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>4</td>
<td>s9</td>
<td>APPLE</td>
</tr>
</tbody>
</table>
The Backpointer Table

• Each entry in the BP table has:
  – An end time
  – An implicit start time
    • End time of parent +1
  – A word identity
  – A node score
    • Total score to node – total score to parent
    • Node score may be further separated into
      – Acoustic score
      – Language score
      – Must keep track of acoustic and language model scores separately for this
A different view of the BP table

Each rectangle is a BP table entry, with a start time, an end time, a word id, and a score. Some entries have no children.
The BP Table as a Tree

- **Introduce the begin-utterance and end-utterance markers**
- **Note**: Each node has a score
  - Acoustic score and LM score
  - Can be separated; Acoustic score stays at node, LM score rides incoming edge
The BP Table as a DAG

- Add additional edges
  - Only between nodes whose timestamps match up
  - End frame of one node is immediately before first frame of next
  - New edges can be assigned appropriate LM score
    - Or may be assigned a score via reasonable heuristics
The BP Table as a **DAG**

- “Approximate” edges
  - Add edges between nodes if they are only “slightly” misaligned
  - i.e. have a gap of less than $X$ frames, or overlap by $Y$ frames
  - Typical values: $X = Y = 2$. 
The **LATTICE**

- The resulting structure is a DAG called the “LATTICE”
- It represents the set of all major word-sequence hypotheses that were “considered” by the recognizer
- It includes the final most-probable (best match) word sequence that was obtained, but also much more..
- Note: It’s a probabilistic structure
  - Each node and edge contain probability (or log prob) scores
Returning to ASR

• We now apply what we have learned to address some problems in speech recognition

• N-best generation
• Rescoring
• Confidence estimation

• We will perform all of this using the recognition lattice
Problem 1: N-best hypotheses

• The recognizer always outputs the best-scoring word sequence hypothesis
• What is the second-best scoring hypothesis?
• The third-best scoring hypothesis?
• The Nth-best scoring hypothesis?

• The N-best output procedure
The Stack Decoder for N-best hypotheses

• Apply the stack decoding algorithm to obtain the N-best paths from <s> to </s>
  – Assuming single source node: <s>
  – Assuming one or more sink nodes: </s>
The A* Decoder for N-best hypotheses

- Generally preferable to use the A* algorithm instead of the stack decoder
  - For reasons explained earlier
**Rescoring:** Using a different LM

- **Common tactic:**
  - Perform first pass of recognition with a “simple” language model
    - E.g. a bigram LM
    - Much more compact graph, much more efficient search
  - Find the best path through lattice using a higher-order (or more detailed) LM
  - Also called *Rescoring*

- Easily performed using a modification of the stack or A* decoder
Rescoring through A*

To extend a path:
- Append one-step extensions to current path
- Factor in cost of one-step extensions

The edge costs carry LM probabilities

Refer to word history and obtain LM probabilities from new LM
- Edge score = \( P(W(\text{node}) \mid \text{Word history}) \)**!
Confidence: How sure are we

• How sure can we be that the recognizer output is correct?
  – Often critical to know
  – If we are not confident, we must take corrective action

• No really robust method to compute confidence

• Confidence is often obtained as the *a posteriori* probability
**Confidence as a posteriori probability**

- Any word in our hypothesis represents a node in the lattice.
- The confidence assigned to the word is the *a posteriori probability* of the node:
  - A number between 0 and 1
  - 0 → sure that it's wrong; 1 → sure that it's correct
  - Caveat: We can be wrong when we're sure.
Computing a posteriori probability

• A posteriori probability:
  – Total probability of all paths through node / total probability of graph

• We already know how to compute these
Assigning confidences

- For each node representing word in hypothesis:
  Compute total probability of all paths through node
  - Using forward-backward algorithm given earlier
- Compute total probability of graph
  - Also using the forward-backward algorithm
Assigning confidences

- For each node representing word in hypothesis:
  - Confidence = total prob of paths through node / total prob

- Can in fact be computed for every node in the lattice
Topics covered

• N-best generation
• Rescoring
• Confidence estimation

• Using:
  – A*
    • Combines Dijkstra’s algorithm and stack decoding
  – Forward-Backward algorithm
Additional topics

• Topics remaining:
• Improved confidence scoring
• Acoustic rescoring
• Adaptation
• Neural network methods