Genetic Algorithms for Syntactic Parsing

Sergio Penkale

July 24th, 2008
What is a Genetic Algorithm?

- Heuristic search algorithm
- Based on analogy to biological evolution

1. We randomly build an *Initial Population*
2. While finalisation criteria doesn’t hold:
   1. A percentage of individuals is selected
   2. Individuals are recombinated
   3. Mutations are introduced
3. The individual with the greatest fitness is returned
What is a Genetic Algorithm?

- Heuristic search algorithm
- Based on analogy to biological evolution

1. We randomly build an *Initial Population*
2. While finalisation criteria doesn’t hold:
   1. A percentage of individuals is *selected*
   2. Individuals are *recombined*
   3. *Mutations* are introduced
3. the individual with the greatest *fitness* is returned
Obtaining the initial population

- We will use a traditional parser and ask it to return the best $k$ trees according to its model.
- This way we will obtain a set of possible parses of great quality.
Mutation Operation (Mutations 1 and 2)

- It models a reattachment

Example: Original

Example: Mutation

- The inverse operation is also allowed
It models a reattachment

The inverse operation is also allowed
Mutation Operation (Mutations 1 and 2)

- It models a *reattachment*

**Example: Original**

```
A
  B
  D
  ...  E  F
  ...  ...  ...
```

**Example: Mutation**

```
A
  C
  B
  D
  ...  E  F
  ...  ...  ...
```

- The inverse operation is also allowed
Mutation Operation (Mutations 1 and 2)

- It models a reattachment

Example: Original

```
  A
 / \    
B   C
  / \  
D   E
  / \  
F...
```

Example: Mutation

```
  A
 /     
B   C
  /     
D   E
  /     
F...
```

- The inverse operation is also allowed
Mutation Operation (Mutations 1 and 2)

- It models a *reattachment*

Example: Original

```
  A
 /\  \
B   C
```

Example: Mutation

```
  A
 /\  \
B   C
```

- The inverse operation is also allowed
Mutation Operation (Mutations 1 and 2)

- It models a *reattachment*

Example: Original

```
A
  B
  C
  D
  E
  F
```

Example: Mutation

```
A
  B
  C
  D
  E
  F
```

- The inverse operation is also allowed
Mutations 3/4: Eliminating/Creating nodes

The inverse operation is also allowed.
Mutations 3/4: Eliminating/Creating nodes

The inverse operation is also allowed
Mutations 3/4: Eliminating/Creating nodes

Original:

```
VP
  VBN
  Open
  NP
    NP
      DT
      NN
      the
      door
    PP
      with the key
```

Mutation:

```
VP
  VBN
  Open
  NP
    DP
      DT
      NN
      the
      door
    PP
      with the key
```

- The inverse operation is also allowed
Mutations 3/4: Eliminating/Creating nodes

- The inverse operation is also allowed
Mutations 3/4: Eliminating/Creating nodes

The inverse operation is also allowed
Mutations 3/4: Eliminating/Creating nodes

- The inverse operation is also allowed
Mutations 3/4: Eliminating/Creating nodes

The inverse operation is also allowed

Original

Mutation

- The inverse operation is also allowed
Mutation 5: Upwards reattach creating node

Original

Mutation

Sergio Penkale
Genetic Algorithms for Syntactic Parsing
Mutation 5: Upwards reattach creating node

Original

```
Original
<table>
<thead>
<tr>
<th>VBN</th>
<th>NP</th>
<th>PP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>DT, the, NN</td>
<td>with the key</td>
</tr>
</tbody>
</table>
```
The crossover operation

- It takes two “parents” syntactic trees
- Subfragments are recombined
- It returns two “child” syntactic trees
The crossover operation

- It takes two “parents” syntactic trees
- Subfragments are recombined
- It returns two “child” syntactic trees
The crossover operation

- It takes two “parents” syntactic trees
- Subfragments are recombined
- It returns two “child” syntactic trees
Crossover: example

Parent 1

S

NP1

VP1

...

Parent 2

S

NP2

VP2

...

Child 1

S

NP2

VP1

...

Child 2

S

NP1

VP2

...
Crossover: example

**Parent 1**

```
S
  NP1  VP1
    ...  ...
```

**Parent 2**

```
S
  NP2  VP2
    ...  ...
```

**Child 1**

```
S
  NP2  VP1
    ...  ...
```

**Child 2**

```
S
  NP1  VP2
    ...  ...
```
The Fitness Function

- Quantifies how well an individual adapts to its environment
- Determines the probability for selecting individuals
- Determines which features from the individuals the GA will try to maximise
- We will define a few fitness functions and will evaluate their influence in the behaviour of the algorithm
The Fitness Function

- Quantifies how well an individual adapts to its environment
- Determines the probability for selecting individuals
- Determines which features from the individuals the GA will try to maximise
- We will define a few fitness functions and will evaluate their influence in the behaviour of the algorithm
The Fitness Function

- Quantifies how well an individual adapts to its environment
- Determines the probability for selecting individuals
- Determines which features from the individuals the GA will try to maximise

We will define a few fitness functions and will evaluate their influence in the behaviour of the algorithm.
The Fitness Function

- Quantifies how well an individual adapts to its environment
- Determines the probability for selecting individuals
- Determines which features from the individuals the GA will try to maximise
- We will define a few fitness functions and will evaluate their influence in the behaviour of the algorithm
We will define a fitness function that cheats: CheaterFitness

- It has access to the reference tree created by a human

\[
\text{CheaterFitness} = \text{F-Measure}
\]

- It maximises exactly the criteria we will use to evaluate
- It’s purpose is to verify that the defined operators have sufficient expressive power
We will define a fitness function that cheats: CheaterFitness

It has access to the reference tree created by a human

\[
\text{CheaterFitness} = \text{F-Measure}
\]

It maximises exactly the criteria we will use to evaluate

It’s purpose is to verify that the defined operators have sufficient expressive power
We will define a fitness function that cheats: CheaterFitness

- It has access to the reference tree created by a human
- CheaterFitness = F-Measure

- It maximises exactly the criteria we will use to evaluate
- Its purpose is to verify that the defined operators have sufficient expressive power
We will define a fitness function that cheats: CheaterFitness

- It has access to the reference tree created by a human

\[
\text{CheaterFitness} = \text{F-Measure}
\]

- It maximises exactly the criteria we will use to evaluate
- Its purpose is to verify that the defined operators have sufficient expressive power
We will define a fitness function that cheats: CheaterFitness

- It has access to the reference tree created by a human

\[
\text{CheaterFitness} = \text{F-Measure}
\]

- It maximises exactly the criteria we will use to evaluate
- It’s purpose is to verify that the defined operators have sufficient expressive power
We will define a fitness function that assigns high values to trees whose structures are “typical”

This is determined by subtrees contained in a set of chunks, built during training

We add the lengths of the spans of subtrees contained in chunks

We divide by the sum of the lengths of the spans of all subtrees

\[
\text{ChunkFitness}(\tau) = \frac{\sum_{\{\rho \in ST(\tau): \rho \in \text{chunks}\}} \text{len}(\rho)}{\sum_{\{\rho \in ST(\tau)\}} \text{len}(\rho)}
\]
We will define a fitness function that assigns high values to trees whose structures are “typical”

This is determined by subtrees contained in a set $\text{chunks}$, built during training

We add the lengths of the spans of subtrees contained in $\text{chunks}$

We divide by the sum of the lengths of the spans of all subtrees

$$
\text{ChunkFitness}(\tau) = \frac{\sum_{\rho \in ST(\tau): \rho \in \text{chunks}} \text{len}(\rho)}{\sum_{\rho \in ST(\tau)} \text{len}(\rho)}
$$
We will define a fitness function that assigns high values to trees whose structures are “typical”.

This is determined by subtrees contained in a set \( \text{chunks} \), built during training.

We add the lengths of the spans of subtrees contained in \( \text{chunks} \).

We divide by the sum of the lengths of the spans of all subtrees.

\[
\text{ChunkFitness}(\tau) = \frac{\sum_{\rho \in \text{ST}(\tau): \rho \in \text{chunks}} \text{len}(\rho)}{\sum_{\rho \in \text{ST}(\tau)} \text{len}(\rho)}
\]
We will define a fitness function that assigns high values to trees whose structures are “typical”

This is determined by subtrees contained in a set $\text{chunks}$, built during training

We add the lengths of the spans of subtrees contained in $\text{chunks}$

We divide by the sum of the lengths of the spans of all subtrees

$$\text{ChunkFitness}(\tau) = \frac{\sum_{\{\rho \in \mathcal{ST}(\tau): \rho \in \text{chunks}\}} \text{len}(\rho)}{\sum_{\{\rho \in \mathcal{ST}(\tau)\}} \text{len}(\rho)}$$
The ChunkFitness function

- We will define a fitness function that assigns high values to trees whose structures are “typical”
- This is determined by subtrees contained in a set $\text{chunks}$, built during training
- We add the lengths of the spans of subtrees contained in $\text{chunks}$
- We divide by the sum of the lengths of the spans of all subtrees

$$\text{ChunkFitness}(\tau) = \frac{\sum\{\rho \in ST(\tau): \rho \in \text{chunks}\} \text{len}(\rho)}{\sum\{\rho \in ST(\tau)\} \text{len}(\rho)}$$
The PCFGFitness Function

- We obtain the PCFG that covers the treebank
- We smooth the probabilities of unseen rules with the function:

  \[ \text{unseen}(A) = \text{lowest probability among rules with } A \text{ on the left side} \]

Then

\[ \text{PCFGFitness}(\tau) = 1 - \frac{\log_{10}(\prod_{r \in \text{rules}(\tau)} \Theta(r))}{\log_{10}(\prod_{r \in \text{rules}(\tau)} \text{unseen}(r))} \]
The PCFGFitness Function

- We obtain the PCFG that covers the treebank
- We smooth the probabilities of unseen rules with the function:

\[
\text{unseen}(A) = \text{lowest probability among rules with } A \text{ on the left side}
\]

Then

\[
\text{PCFGFitness}(\tau) = 1 - \frac{\log_{10}(\prod_{r \in \text{rules}(\tau)} \Theta(r))}{\log_{10}(\prod_{r \in \text{rules}(\tau)} \text{unseen}(r))}
\]
The PCFGFitness Function

- We obtain the PCFG that covers the treebank
- We smooth the probabilities of unseen rules with the function:

\[
\text{unseen}(A) = \text{lowest probability among rules with } A \text{ on the left side}
\]

Then

\[
\text{PCFGFitness}(\tau) = 1 - \frac{\log_{10}(\prod \{r \in \text{rules}(\tau)\} \Theta(r))}{\log_{10}(\prod \{r \in \text{rules}(\tau)\} \text{unseen}(r))}
\]
We defined a few grammatical rules we wish the tree satisfy

- They are not fitness functions. They modify values returned by fitness functions
- They don’t use lexical information
We defined a few grammatical rules we wish the tree satisfy.

They are not fitness functions. They modify values returned by fitness functions.

They don’t use lexical information.
We defined a few grammatical rules we wish the tree satisfy

- They are not fitness functions. They modify values returned by fitness functions
- They don’t use lexical information
We made several experiments to evaluate the behaviour of each fitness function.

- **We used the Wall Street Journal section of the Penn Treebank.**
- We used sections 02-21 for training and evaluated using section 22.
- For initial populations we used Bikel’s parser with settings set to emulate Collins ’99.
We made several experiments to evaluate the behaviour of each fitness function

- We used the Wall Street Journal section of the Penn Treebank
- We used sections 02-21 for training and evaluated using section 22
- For initial populations we used Bikel’s parser with settings set to emulate Collins ’99
We made several experiments to evaluate the behaviour of each fitness function

- We used the Wall Street Journal section of the Penn Treebank
- We used sections 02-21 for training and evaluated using section 22
- For initial populations we used Bikel’s parser with settings set to emulate Collins ’99
## Results

<table>
<thead>
<tr>
<th>Sentences with 1 to 35 words</th>
<th>PREC</th>
<th>REC</th>
<th>F-Measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>CheaterFitness</td>
<td>99.51</td>
<td>99.02</td>
<td>99.19</td>
</tr>
<tr>
<td>PCFGFitness w/GR</td>
<td>74.73</td>
<td>83.77</td>
<td>78.63</td>
</tr>
<tr>
<td>PCFGFitness</td>
<td>74.36</td>
<td>83.82</td>
<td>78.43</td>
</tr>
<tr>
<td>ChunkFitness w/GR</td>
<td>70.97</td>
<td>67.03</td>
<td>68.43</td>
</tr>
<tr>
<td>ChunkFitness</td>
<td>71.03</td>
<td>66.74</td>
<td>68.29</td>
</tr>
</tbody>
</table>

Sergio Penkale  
Genetic Algorithms for Syntactic Parsing
Conclusions

- The excellent behaviour of the cheater fitness shows that the defined operations have the necessary expressive power to achieve good results.
- The implementations made during this work can be used as a framework to investigate fitness functions in future work.
- The inclusion of a few grammatical rules improved the results.
Thank you