Machine Learning applied to Go

Dmitry Kamenetsky

Supervisor: Nic Schraudolph

March 2007
1. Introduction

2. Monte Carlo Go

3. My Work
What is Go?

- Two-player deterministic board game
- Originated in ancient China. Today very popular in China, Japan and Korea
- 19x19 grid, also 9x9 grid for beginners
- Simple rules, but very complex strategy
Introduction
Monte Carlo Go
My Work

Why study Go?

- *If there are sentient beings on other planets, then they play Go*
  – Emanuel Lasker, former chess world champion

- *Go is one of the grand challenges of AI*
  – Ron Rivest, professor of Computer Science at MIT

- *Go is like life and life is like Go*
  – Chinese proverb
Why study Go?

- *If there are sentient beings on other planets, then they play Go*
  – Emanuel Lasker, former chess world champion

- *Go is one of the grand challenges of AI*
  – Ron Rivest, professor of Computer Science at MIT

- *Go is like life and life is like Go*
  - Chinese proverb
Why study Go?

- *If there are sentient beings on other planets, then they play Go*  
  – Emanuel Lasker, former chess world champion

- *Go is one of the grand challenges of AI*  
  – Ron Rivest, professor of Computer Science at MIT

- *Go is like life and life is like Go*  
  - Chinese proverb
Monte Carlo

- Often used for problems that have no closed solution, e.g. computational physics
- Sample instances from some large population
- Use samples to approximate some common property of the population
- In search, select the next state based on some fixed distribution (usually uniform)
- Recently, have been very successful in Go, causing a mini-revolution
Monte Carlo

- Often used for problems that have no closed solution, e.g. computational physics
- Sample instances from some large population
  - Use samples to approximate some common property of the population
  - In search, select the next state based on some fixed distribution (usually uniform)
  - Recently, have been very successful in Go, causing a mini-revolution
Monte Carlo

- Often used for problems that have no closed solution, e.g. computational physics
- Sample instances from some large population
- Use samples to approximate some common property of the population
- In search, select the next state based on some fixed distribution (usually uniform)
- Recently, have been very successful in Go, causing a mini-revolution
Monte Carlo

- Often used for problems that have no closed solution, e.g. computational physics
- Sample instances from some large population
- Use samples to approximate some common property of the population
- In search, select the next state based on some fixed distribution (usually uniform)
- Recently, have been very successful in Go, causing a mini-revolution
Monte Carlo

- Often used for problems that have no closed solution, e.g. computational physics

- Sample instances from some large population

- Use samples to approximate some common property of the population

- In search, select the next state based on some fixed distribution (usually uniform)

- Recently, have been very successful in Go, causing a mini-revolution
K-armed bandit problem

- Slot machine with K arms. Each arm provides a reward based on some unknown, but fixed distribution

- Goal: to choose arms to play such that the total reward is maximized

- How should the gambler play at any given moment?
  - Choose arm with highest average reward seen so far (exploitation)
  - Choose a sub-optimal arm in the hope that it will lead to a greater reward (exploration)
  - Neither - need a combination of exploitation and exploration
K-armed bandit problem

- Slot machine with K arms. Each arm provides a reward based on some unknown, but fixed distribution

- Goal: to choose arms to play such that the total reward is maximized

- How should the gambler play at any given moment?
  - Choose arm with highest average reward seen so far (exploitation)
  - Choose a sub-optimal arm in the hope that it will lead to a greater reward (exploration)
  - Neither – need a combination of exploitation and exploration
K-armed bandit problem

- Slot machine with K arms. Each arm provides a reward based on some unknown, but fixed distribution.

- Goal: to choose arms to play such that the total reward is maximized.

- How should the gambler play at any given moment?
  - Choose arm with highest average reward seen so far (exploitation).
  - Choose a sub-optimal arm in the hope that it will lead to a greater reward (exploration).
  - Neither - need a combination of exploitation and exploration.
K-armed bandit problem

- Slot machine with K arms. Each arm provides a reward based on some unknown, but fixed distribution.

- Goal: to choose arms to play such that the total reward is maximized.

- How should the gambler play at any given moment?
  - Choose arm with highest average reward seen so far (exploitation).
  - Choose a sub-optimal arm in the hope that it will lead to a greater reward (exploration).
  - Neither - need a combination of exploitation and exploration.
K-armed bandit problem

- Slot machine with K arms. Each arm provides a reward based on some unknown, but fixed distribution.

- Goal: to choose arms to play such that the total reward is maximized.

- How should the gambler play at any given moment?
  - Choose arm with highest average reward seen so far (exploitation).
  - Choose a sub-optimal arm in the hope that it will lead to a greater reward (exploration).
  - Neither - need a combination of exploitation and exploration.
K-armed bandit problem

- Slot machine with K arms. Each arm provides a reward based on some unknown, but fixed distribution.

- Goal: to choose arms to play such that the total reward is maximized.

- How should the gambler play at any given moment?
  - Choose arm with highest average reward seen so far (exploitation).
  - Choose a sub-optimal arm in the hope that it will lead to a greater reward (exploration).
  - Neither - need a combination of exploitation and exploration.
Upper Confidence Bounds

- Successive plays of arm \( i \) give rewards \( X_{i,1}, X_{i,2}, ... \) which are i.i.d. with unknown \( \mathbb{E}(X_i) = \mu_i \)

- Let \( T_i(n) \) be the number of times arm \( i \) has been played during the first \( n \) plays of machine

- Upper Confidence Bounds - UCB (Auer et al. 2002)
  - Initialization: Play each arm once
  - Loop: Play arm \( i \) that maximizes \( \mu_i + \sqrt{\frac{2 \log n}{T_i(n)}} \)
  - Proven to achieve optimal regret
Upper Confidence Bounds

- Successive plays of arm $i$ give rewards $X_{i,1}, X_{i,2}, \ldots$ which are i.i.d. with unknown $\mathbb{E}(X_i) = \mu_i$

- Let $T_i(n)$ be the number of times arm $i$ has been played during the first $n$ plays of machine

- Upper Confidence Bounds - UCB (Auer et al. 2002)
  - Initialization: Play each arm once
  - Loop: Play arm $i$ that maximizes $\mu_i + \sqrt{\frac{2 \log n}{T_i(n)}}$
  - Proven to achieve optimal regret
Upper Confidence Bounds

- Successive plays of arm $i$ give rewards $X_{i,1}, X_{i,2}, \ldots$ which are i.i.d. with unknown $\mathbb{E}(X_i) = \mu_i$

- Let $T_i(n)$ be the number of times arm $i$ has been played during the first $n$ plays of machine

- Upper Confidence Bounds - UCB (Auer et al. 2002)
  - Initialization: Play each arm once
  - Loop: Play arm $i$ that maximizes $\mu_i + \sqrt{\frac{2 \log n}{T_i(n)}}$
  - Proven to achieve optimal regret
Upper Confidence Bounds

- Successive plays of arm $i$ give rewards $X_{i,1}, X_{i,2}, ...$ which are i.i.d. with unknown $\mathbb{E}(X_i) = \mu_i$

- Let $T_i(n)$ be the number of times arm $i$ has been played during the first $n$ plays of machine

Upper Confidence Bounds - UCB (Auer et al. 2002)

  - Initialization: Play each arm once

    - Loop: Play arm $i$ that maximizes $\mu_i + \sqrt{\frac{2 \log n}{T_i(n)}}$

  - Proven to achieve optimal regret
Upper Confidence Bounds

- Successive plays of arm $i$ give rewards $X_{i,1}, X_{i,2}, \ldots$ which are i.i.d. with unknown $\mathbb{E}(X_i) = \mu_i$

- Let $T_i(n)$ be the number of times arm $i$ has been played during the first $n$ plays of machine

Upper Confidence Bounds - UCB (Auer et al. 2002)

- Initialization: Play each arm once

- Loop: Play arm $i$ that maximizes $\mu_i + \sqrt{\frac{2\log n}{T_i(n)}}$

Proven to achieve optimal regret
Upper Confidence Bounds

- Successive plays of arm $i$ give rewards $X_{i,1}, X_{i,2}, \ldots$ which are i.i.d. with unknown $\mathbb{E}(X_i) = \mu_i$

- Let $T_i(n)$ be the number of times arm $i$ has been played during the first $n$ plays of machine

- Upper Confidence Bounds - UCB (Auer et al. 2002)
  - Initialization: Play each arm once
  - Loop: Play arm $i$ that maximizes $\mu_i + \sqrt{\frac{2 \log n}{T_i(n)}}$

- Proven to achieve optimal regret
UCT

- UCB for minimax tree search (Kocsis and Szepesvari 2006)

- Start at the current board position $p$

- For $i = 1$ to 100,000 (number of simulations)
  - $p' \leftarrow p$
    - until stopping criterion is reached (e.g., end of game)

  Evaluate leaf; value $\leftarrow$ winner of $p'$
  Update all the visited nodes with value

- At $p$ play move with highest winning percentage
**UCT**

- UCB for minimax tree search (Kocsis and Szepesvari 2006)

- Start at the current board position $p$

  - For $i = 1$ to 100,000 (number of simulations)
    - $p' \leftarrow p$
    - until stopping criterion is reached (e.g. end of game)

- Evaluate leaf: value $\leftarrow$ winner of $p'$

- Update all the visited nodes with value

- At $p$ play move with highest winning percentage
UCB for minimax tree search (Kocsis and Szepesvari 2006)

Start at the current board position $p$

For $i = 1$ to 100,000 (number of simulations)

- $p' \leftarrow p$
- until stopping criterion is reached (e.g. end of game)
  - $p'' \leftarrow p' + \text{move given by UCB}$
  - create node $p''$
- Evaluate leaf: value $\leftarrow \text{winner of } p''$
- Update all the visited nodes with value

At $p$ play move with highest winning percentage
UCT

- UCB for minimax tree search (Kocsis and Szepesvari 2006)
- Start at the current board position $p$
- For $i = 1$ to 100,000 (number of simulations)
  - $p' \leftarrow p$
  - until stopping criterion is reached (e.g. end of game)
    - $p' \leftarrow p' + \text{move given by UCB}$
    - create node $p'$
  - Evaluate leaf: value $\leftarrow$ winner of $p'$
  - Update all the visited nodes with value
- At $p$ play move with highest winning percentage
UCT

- UCB for minimax tree search (Kocsis and Szepesvari 2006)
- Start at the current board position p
- For i = 1 to 100,000 (number of simulations)
  - $p' \leftarrow p$
  - until stopping criterion is reached (e.g. end of game)
    - $p' \leftarrow p' + \text{move given by UCB}$
    - create node $p'$
    - Evaluate leaf: value $\leftarrow \text{winner of } p'$
    - Update all the visited nodes with value
- At $p$ play move with highest winning percentage
UCB for minimax tree search (Kocsis and Szepesvari 2006)

Start at the current board position $p$

For $i = 1$ to 100,000 (number of simulations)
  
  $p' \leftarrow p$
  
  until stopping criterion is reached (e.g. end of game)
    
    $p' \leftarrow p' + \text{move given by UCB}$
    
    create node $p'$
  
  Evaluate leaf: value $\leftarrow \text{winner of } p'$
  
  Update all the visited nodes with value

At $p$ play move with highest winning percentage
Monte Carlo Go

My Work

Monte Carlo Bandit Problem

MoGo

Introduction

UCT

- UCB for minimax tree search (Kocsis and Szepesvari 2006)
- Start at the current board position $p$
- For $i = 1$ to 100,000 (number of simulations)
  - $p' \leftarrow p$
  - until stopping criterion is reached (e.g. end of game)
    - $p' \leftarrow p' +$ move given by UCB
    - create node $p'$
    - Evaluate leaf: value $\leftarrow$ winner of $p'$
    - Update all the visited nodes with value
- At $p$ play move with highest winning percentage

Dmitry Kamenetsky

Machine Learning applied to Go
**UCB**

- UCB for minimax tree search (Kocsis and Szepesvari 2006)
- Start at the current board position \( p \)
- For \( i = 1 \) to 100,000 (number of simulations)
  - \( p' \leftarrow p \)
  - until stopping criterion is reached (e.g. end of game)
    - \( p' \leftarrow p' + \) move given by UCB
    - create node \( p' \)
  - Evaluate leaf: value \( \leftarrow \) winner of \( p' \)
  - Update all the visited nodes with value
- At \( p \) play move with highest winning percentage
Introduction

Monte Carlo Go

My Work

Monte Carlo Bandit Problem

MoGo

UCT

- UCB for minimax tree search (Kocsis and Szepesvari 2006)
- Start at the current board position $p$
- For $i = 1$ to 100,000 (number of simulations)
  - $p' \leftarrow p$
  - until stopping criterion is reached (e.g. end of game)
    - $p' \leftarrow p' +$ move given by UCB
    - create node $p'$
  - Evaluate leaf: value $\leftarrow$ winner of $p'$
  - Update all the visited nodes with value
- At $p$ play move with highest winning percentage
**UCB for minimax tree search** (Kocsis and Szepesvari 2006)

- Start at the current board position \( p \)

- For \( i = 1 \) to 100,000 (number of simulations)
  - \( p' \leftarrow p \)
  - until stopping criterion is reached (e.g. end of game)
    - \( p' \leftarrow p' + \) move given by UCB
    - create node \( p' \)
  - Evaluate leaf: value \( \leftarrow \) winner of \( p' \)
  - Update all the visited nodes with value

- At \( p \) play move with highest winning percentage
MoGo

- First Go program to use UCT (Gelly et. al 2006)
- Store nodes and their statistics in a tree data structure
- Stopping criterion is a node that is not yet in the tree
- Leaf node evaluation:
  - Pruning techniques, smart ordering of unexplored moves
MoGo

- First Go program to use UCT (Gelly et. al 2006)
- Store nodes and their statistics in a tree data structure
- Stopping criterion is a node that is not yet in the tree
- Leaf node evaluation:
  - Playout position randomly until no moves remain. Final position is trivial to score
  - Enhanced through the use of patterns and playing near the previous move
- Pruning techniques, smart ordering of unexplored moves
MoGo

- First Go program to use UCT (Gelly et. al 2006)
- Store nodes and their statistics in a tree data structure
- Stopping criterion is a node that is not yet in the tree
  - Leaf node evaluation:
    - Playout position randomly until no moves remain. Final position is trivial to score
    - Enhanced through the use of patterns and playing near the previous move
  - Pruning techniques, smart ordering of unexplored moves
First Go program to use UCT (Gelly et. al 2006)

- Store nodes and their statistics in a tree data structure
- Stopping criterion is a node that is not yet in the tree

Leaf node evaluation:
- Playout position randomly until no moves remain. Final position is trivial to score
- Enhanced through the use of patterns and playing near the previous move
- Pruning techniques, smart ordering of unexplored moves
MoGo

- First Go program to use UCT (Gelly et. al 2006)
- Store nodes and their statistics in a tree data structure
- Stopping criterion is a node that is not yet in the tree

Leaf node evaluation:

- Playout position randomly until no moves remain. Final position is trivial to score
  - Enhanced through the use of patterns and playing near the previous move

- Pruning techniques, smart ordering of unexplored moves
MoGo

- First Go program to use UCT (Gelly et. al 2006)
- Store nodes and their statistics in a tree data structure
- Stopping criterion is a node that is not yet in the tree
- Leaf node evaluation:
  - Playout position randomly until no moves remain. Final position is trivial to score
  - Enhanced through the use of patterns and playing near the previous move
- Pruning techniques, smart ordering of unexplored moves
MoGo

- First Go program to use UCT (Gelly et. al 2006)
- Store nodes and their statistics in a tree data structure
- Stopping criterion is a node that is not yet in the tree
- Leaf node evaluation:
  - Playout position randomly until no moves remain. Final position is trivial to score
  - Enhanced through the use of patterns and playing near the previous move
- Pruning techniques, smart ordering of unexplored moves
MoGo’s success

- Ranked first on 9x9 Computer Go Server since August 2006
- Won two most recent tournaments on 9x9 and 13x13
- Expected to reach the level of human professional on 9x9 board
MoGo’s success

- Ranked first on 9x9 Computer Go Server since August 2006
- Won two most recent tournaments on 9x9 and 13x13
- Expected to reach the level of human professional on 9x9 board
MoGo’s success

- Ranked first on 9x9 Computer Go Server since August 2006
- Won two most recent tournaments on 9x9 and 13x13
- Expected to reach the level of human professional on 9x9 board
Replacing UCB

- UCT is ad-hoc. Lack of theoretical analysis, because random variables (rewards $X_i$) are not i.i.d.

- Instead, use Beta distributions to model random variables

- Beta distribution is a conjugate prior to binomial distribution (game result)

- Here $\alpha = \text{wins from node}$, $\beta = \text{losses from node}$

- Let $p$ be parent’s winning percentage and $0 < a < 1$ parameter

- Pick a move that is most likely to have a winning percentage greater than $(1 - a)p + a$
Replacing UCB

- UCT is ad-hoc. Lack of theoretical analysis, because random variables (rewards $X_i$) are not i.i.d.

- Instead, use Beta distributions to model random variables

  - Beta distribution is a conjugate prior to binomial distribution (game result)

    - Here $\alpha = \text{wins from node}, \beta = \text{losses from node}$

    - Let $p$ be parent’s winning percentage and $0 < a < 1$ parameter

- Pick a move that is most likely to have a winning percentage greater than $(1 - a)p + a$
Replacing UCB

- UCT is ad-hoc. Lack of theoretical analysis, because random variables (rewards $X_i$) are not i.i.d.

- Instead, use Beta distributions to model random variables

- Beta distribution is a conjugate prior to binomial distribution (game result)

- Here $\alpha$ = wins from node, $\beta$ = losses from node

- Let $p$ be parent’s winning percentage and $0 < a < 1$ parameter

- Pick a move that is most likely to have a winning percentage greater than $(1 - a)p + a$
Replacing UCB

- UCT is ad-hoc. Lack of theoretical analysis, because random variables (rewards $X_i$) are not i.i.d.

- Instead, use Beta distributions to model random variables

- Beta distribution is a conjugate prior to binomial distribution (game result)

- Here $\alpha = \text{wins from node}, \beta = \text{losses from node}$

- Let $p$ be parent’s winning percentage and $0 < a < 1$ parameter

- Pick a move that is most likely to have a winning percentage greater than $(1 - a)p + a$
Re REPLACING UCB

- UCT is ad-hoc. Lack of theoretical analysis, because random variables (rewards $X_i$) are not i.i.d.

- Instead, use Beta distributions to model random variables

- Beta distribution is a conjugate prior to binomial distribution (game result)

- Here $\alpha = \text{wins from node}$, $\beta = \text{losses from node}$

- Let $p$ be parent’s winning percentage and $0 < a < 1$ parameter

- Pick a move that is most likely to have a winning percentage greater than $(1 - a)p + a$
Replacing UCB

- UCT is ad-hoc. Lack of theoretical analysis, because random variables (rewards $X_i$) are not i.i.d.

- Instead, use Beta distributions to model random variables

- Beta distribution is a conjugate prior to binomial distribution (game result)

- Here $\alpha = \text{wins from node}$, $\beta = \text{losses from node}$

- Let $p$ be parent’s winning percentage and $0 < a < 1$ parameter

- Pick a move that is most likely to have a winning percentage greater than $(1 - a)p + a$
MoGo’s node evaluation is fast, but not so meaningful

Instead, use our cooperative scorer:

- Initialization: Statically fill neutral territory with stones
- Loop: players cooperate to make moves that do not affect the score
- Accurately predicts score: 96.3% on 9x9 and 89.2% on 19x19
- Only 15 times slower than pure random
Improving node evaluation

- MoGo’s node evaluation is fast, but not so meaningful
- Instead, use our cooperative scorer:
  - Initialization: Statically fill neutral territory with stones
  - Loop: players cooperate to make moves that do not affect the score
  - Accurately predicts score: 96.3% on 9x9 and 89.2% on 19x19
  - Only 15 times slower than pure random
Improving node evaluation

- MoGo’s node evaluation is fast, but not so meaningful
- Instead, use our cooperative scorer:
  - Initialization: Statically fill neutral territory with stones
  - Loop: players cooperate to make moves that do not affect the score
- Accurately predicts score: 96.3% on 9x9 and 89.2% on 19x19
- Only 15 times slower than pure random
Improving node evaluation

- MoGo’s node evaluation is fast, but not so meaningful
- Instead, use our cooperative scorer:
  - Initialization: Statically fill neutral territory with stones
  - Loop: players cooperate to make moves that do not affect the score
- Accurately predicts score: 96.3% on 9x9 and 89.2% on 19x19
- Only 15 times slower than pure random
MoGo’s node evaluation is fast, but not so meaningful

Instead, use our cooperative scorer:

- Initialization: Statically fill neutral territory with stones

- Loop: players cooperate to make moves that do not affect the score

Accurately predicts score: 96.3% on 9x9 and 89.2% on 19x19

Only 15 times slower than pure random
Improving node evaluation

- MoGo’s node evaluation is fast, but not so meaningful
- Instead, use our cooperative scorer:
  - Initialization: Statically fill neutral territory with stones
  - Loop: players cooperate to make moves that do not affect the score
- Accurately predicts score: 96.3% on 9x9 and 89.2% on 19x19
- Only 15 times slower than pure random
Scorer Example
Scorer Example
Scorer Example
Improving memory management

- Tree data structure is memory-inefficient
- Instead, use a hash table:
  - For each visited position \( p \), key = ZobristHash(\( p \))
  - Store statistics of \( p \): hashTable[\( key \)] = (\#wins, \#runs, depth)
- Collision handling
  - Use a small hash table with more information for frequently visited nodes
Improving memory management

- Tree data structure is memory-inefficient

- Instead, use a hash table:
  - For each visited position $p$, key = ZobristHash($p$)
  - Store statistics of $p$: hashTable[key] = (#wins, #runs, depth)
  - Collision handling

- Use a small hash table with more information for frequently visited nodes
Improving memory management

- Tree data structure is memory-inefficient
- Instead, use a hash table:
  - For each visited position $p$, key = ZobristHash($p$)
  - Store statistics of $p$: hashTable[key] = (#wins, #runs, depth)
  - Collision handling
- Use a small hash table with more information for frequently visited nodes
Improving memory management

- Tree data structure is memory-inefficient
- Instead, use a hash table:
  - For each visited position $p$, key = ZobristHash($p$)
  - Store statistics of $p$: hashTable[key] = (#wins, #runs, depth)
- Collision handling
  - Use a small hash table with more information for frequently visited nodes
Improving memory management

- Tree data structure is memory-inefficient
- Instead, use a hash table:
  - For each visited position $p$, key = ZobristHash($p$)
  - Store statistics of $p$: hashTable[\text{key}] = (wins, runs, depth)
  - Collision handling

- Use a small hash table with more information for frequently visited nodes
Improving memory management

- Tree data structure is memory-inefficient
- Instead, use a hash table:
  - For each visited position $p$, key $= \text{ZobristHash}(p)$
  - Store statistics of $p$: hashTable[key] = (#wins, #runs, depth)
  - Collision handling
- Use a small hash table with more information for frequently visited nodes
Learning evaluation function

- Convert the board position into a graph:
  - Collapse regions of the same colour into one node
  - Create edges between adjacent regions
- Use Condition Random Fields (CRF) to learn from this graph:
- Can use this with the scorer or for move generation
Learning evaluation function

- Convert the board position into a graph:
  - Collapse regions of the same colour into one node
  - Create edges between adjacent regions

- Use Condition Random Fields (CRF) to learn from this graph:
  - Learn final territory assignment
  - Predict the next move

- Can use this with the scorer or for move generation
Learning evaluation function

- Convert the board position into a graph:
  - Collapse regions of the same colour into one node
  - Create edges between adjacent regions

- Use Condition Random Fields (CRF) to learn from this graph:
  - Learn final territory assignment
  - Predict the next move

- Can use this with the scorer or for move generation
Learning evaluation function

- Convert the board position into a graph:
  - Collapse regions of the same colour into one node
  - Create edges between adjacent regions

- Use Condition Random Fields (CRF) to learn from this graph:
  - Learn final territory assignment
  - Predict the next move

- Can use this with the scorer or for move generation
Learning evaluation function

- Convert the board position into a graph:
  - Collapse regions of the same colour into one node
  - Create edges between adjacent regions

- Use Condition Random Fields (CRF) to learn from this graph:
  - Learn final territory assignment
  - Predict the next move

- Can use this with the scorer or for move generation
Learning evaluation function

- Convert the board position into a graph:
  - Collapse regions of the same colour into one node
  - Create edges between adjacent regions

- Use Condition Random Fields (CRF) to learn from this graph:
  - Learn final territory assignment
  - Predict the next move

- Can use this with the scorer or for move generation
Learning evaluation function

- Convert the board position into a graph:
  - Collapse regions of the same colour into one node
  - Create edges between adjacent regions

- Use Condition Random Fields (CRF) to learn from this graph:
  - Learn final territory assignment
  - Predict the next move

- Can use this with the scorer or for move generation
Graph conversion example
Graph conversion example
Questions?

- You never ever know if you never ever ever GO!