Hanson1

导航
博客园
首页
新随笔
联系
订阅XML
管理

< 2017年10月 >

| 日 | 一 | 二 | 三 | 四 | 五 | 六 |
|---|---|---|---|---|---|---|
| 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 | 20 | 21 |
| 22 | 23 | 24 | 25 | 26 | 27 | 28 |
| 29 | 30 | 31 | 1 | 2 | 3 | 4 |





搜索

# 蒙特卡洛树搜索介绍

**与游戏AI有关的问题一般开始于被称作完全信息博弈的游戏。这是一款对弈玩家彼此没有信息可以隐藏的回合制游戏且在游戏技术里没有运气元素(如扔骰子或从洗好的牌中抽牌)，井字过三关，四子棋，跳棋，国际象棋，黑白棋和围棋用到了这个算法的所有游戏。因为在这个游戏类型中发生的任何事件是能够用一棵树完全确定，它能构建所有可能的结果，且能分配一个值用于确定其中一名玩家的赢或输。尽可能找到最优解，然而在树上做一个搜索操作，用选择的方法在每层上交替取最大值和最小值，匹配不同玩家的矛盾冲突目标，顺着这颗树上搜素，这个叫做极小化极大算法。**

*用这个极小化极大算法解决这个问题，完整搜索这颗博弈树花费总时间太多且不切实际。考虑到这类游戏在实战中具有极多的分支因素或每转一圈可移动的高胜率步数，因为这个极小化极大算法需要搜索树中所有节点以便找到最优解且必须检查的节点的数量与分支系数呈指数增长。有解决这个问题的办法，例如仅搜索向前移动(或层)的有限步数且使用评价函数估算出这个位置的胜率，或者如果它们没有价值可以用pruning算法分支。许多这些技术，需要游戏计算机领域的相关知识，可能很难收集到有用的信息。这种方法产生的国际象棋程序能够击败特级大师，类似的成功已经难以实现，特别是19X19的围棋项目。*

*然而，对于高分支的游戏已经有一项游戏AI技术做得很好且占据游戏程序领域的主导地位。这很容易创建一个仅用较小的分支因子就能给出一个较好的游戏结果的算法基本实现，相对简单的修改可以建立和改善它，如象棋或围棋。它能被设置任何游戏规定的时间停止后，用较长的思考时间来学习游戏高手的玩法。因为它不需要游戏的具体知识，它可用于一般的游戏。它甚至可以适应游戏中的随机性规律，这项技术被称为蒙特卡洛树搜索。在这篇文章中我将描述MCTS是如何工作的，特别是一个被称作UCT博弈树搜索的变种，然后将告*

*诉你如何在Python中建立一个基本的实现。*

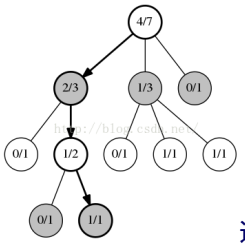
*试想一下，如果你愿意这么做，那么你面临着laohuji的中奖概率,每一个不同的支付概率和金额。作为一个理性的人（如果你要发挥他们的话），你会更喜欢使用的策略，让您能够最大化你的净收益。但是，你怎么能这样做呢？无论出于何种原因附近没有一个人，所以你不能看别人玩一会儿，以获取最好的机器信息，这是最好的机器通过构建统计的置信区间为每台机器做到这一点* $\bar{x}_i \pm \sqrt{\frac{2\ln n}{n_i}}$ *.*

这里：1. $\bar{x}_i$ 平均花费机器i.  2.ni:第i个机器玩家. 3.n:玩家的总数.

然后，你的策略是每次选择机器的最高上界。当你这样做时，因为这台机器所观察到的平均值将改变且它的置信区间会变窄，但所有其它机器的置信区间将扩大。最终，其他机器中将有一个上限超出你的当前机器，你将切换到这台机器。这种策略有你沮丧的性能，你只将在真正的最好的laohuji上玩的不同且根据该策略你将赢得预期奖金，你仅使用O(ln n)时间复杂度。对于这个问题以相同的大O增长率为理论最适合(被称为多臂吃角子老虎问题)，且具有容易计算的额外好处。

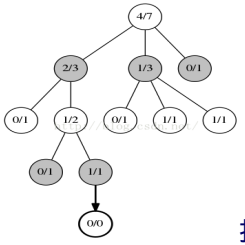
而这里的蒙特卡洛树是怎么来的，在一个标准的蒙特卡罗代码程序中，运用了大量的随机模拟，在这种情况下，从你想找到的最佳移动位置，以起始状态为每个可能的移动做统计，最佳的移动结果被返回。虽然这种移动方法有缺陷，不过，是在用于仿真中任何给定的回合，可能有很多可能的移动，但只有一个或两个是良好。如果每回合随机移动被选择，他将很难发现最佳前进路线。所以，UCT是一个加强算法。我们的想法是这样的，如果统计数据适用于所有仅移动一格的位置，棋盘中的任何位置都可以视为多臂吃角子老虎问题。所以代替许多单纯随机模拟，UCT工作用于许多多阶段淘汰赛。

第一阶段，你有必要持续选择处理每个位置的统计数据时，用来完成一个多拉杆吃角子laohuji问题。此举使用了UCB1算法代替随机选择，且被认为是应用于获取下一个位置。然后选择开始直到你到达一个不是所有的子结点有记录数据的位置。




选择：此处通过在每一步UCB1算法所选的位置并移动标记为粗体。注意一些玩家间的对弈记录已经被统计下来。每个圆圈中包含玩家的胜场数/次数。

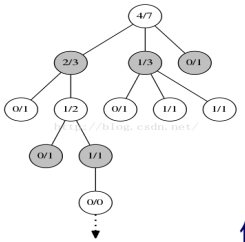
第二阶段，扩容，发现此时已不再适用于UCB1算法。一个未访问的子结点被随机选择，并且记录一个新节点被添加到统计树。




扩张：记录为1/1的位置位于树的底部在它之下没有进一步的统计记录，所以我们选择一个随机移动，并为它添加一个新结点(粗体)，初始化为0/0。

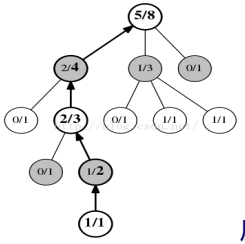
4. Linux根目录各个文件夹介绍及说明 (367)
5. 有了 itchat， python 调用微信个人号从未如此简单（新增 py3 支持）(296)

扩容后，其余部分的开销是在第二阶段，仿真。这么做是经典的蒙特卡洛模拟，或纯随机或如果选择一个年轻选手，则用一些简单的加权探索法，或对于高端玩家，则用一些计算复杂的启发式和估算。对于较小的分支因子游戏，一个年轻选手能给出好的结果。




仿真：一旦新结点被添加，蒙特卡洛模拟开始，这里用虚线箭头描述。模拟移动可以是完全随机的，或可以使用计算加权随机数来取代移动，可能获得更好的随机性。

最后，第四阶段是更新和反转，当比赛结束后，这种情况会发生。所有玩家访问过的位置，其比赛次数递增，如果那个位置的玩家赢得比赛，其胜场递增。




反转：在仿真结束后，所有结点路径被更新。每个人玩一次就递增1，并且每次匹配的获胜者，其赢得游戏次数递增1，这里用粗体字表示。

该算法可以被设置任何期望时间后停止，或在某些其他条件。随着越来越多的比赛进行，博弈树在内存中成长，这个移动将是最终选择，此举将趋近实际的最佳玩法虽然可能需要非常长的时间。

有关UCB1和UCT算法的数学知识，更多详情请看 Finite-time Analysis of the Multiarmed Bandit Problem and Bandit based Monte-Carlo Planning.

现在让我们看看这个AI算法。要分开考虑，我们将需要一个模板类，其目的是封装一个比赛规则且不用关心AI，和一个仅注重于AI算法的蒙特卡洛类，且查询到模板对象以获得有关游戏信息。让我们假设一个模板类支持这个接口：

```
class Board(object):
    def start(self):
        # 表示返回游戏的初始状态。
        pass

    def current_player(self, state):
        # 获取游戏状态并返回当前玩家编号
        pass
```

```
        pass

    def next_state(self, state, play):
        # 获取比赛状态,且这个移动被应用.
        # 返回新的游戏状态.
        pass

    def legal_plays(self, state_history):
        # 采取代表完整的游戏历史记录的游戏状态的序列，且返回当前玩家的合法玩法的完整移动列表。
        pass

    def winner(self, state_history):
        # <span style="font-family: Arial, Helvetica, sans-serif;">采取代表完整的游戏历史记录的游戏状态的序列。</span>
        # 如果现在游戏赢了, 返回玩家编号。
        # 如果游戏仍然继续，返回0。
        # 如果游戏打结, 返回明显不同的值, 例如: -1.
        pass
```

对于这篇文章的目的，我不会给任何进一步详细说明，但对于示例，你可以在github上找到实现代码。不过，需要注意的是，我们需要状态数据结构是哈希表和同等状态返回相同的哈希值是非常重要的。我个人使用平板元组作为我的状态数据结构。

我们将构建能够支持这个接口的人工智能类：

```
class MonteCarlo(object):
    def __init__(self, board, **kwargs):
        # 取一个模板的实例且任选一些关键字参数。
        # 初始化游戏状态和统计数据表的列表。
        pass

    def update(self, state):
        # 需要比赛状态，并追加到历史记录
        pass

    def get_play(self):
        # 根据当前比赛状态计算AI的最佳移动并返回。
        pass

    def run_simulation(self):
        # 从当前位置完成一个“随机”游戏,
        # 然后更新统计结果表
```

```
        # 然后更新统计结果表.
        pass
```

让我们从初始化和保存数据开始。这个AI的模板对象将用于获取有关这个游戏在哪里运行且AI被允许怎么做的信息，所以我们需要将它保存。此外，我们需要保持跟踪数据状态，以便我们获取它。

```
class MonteCarlo(object):
    def __init__(self, board, **kwargs):
        self.board = board
        self.states = []

    def update(self, state):
        self.states.append(state)
```

该UCT算法依赖于当前状态运行的多款游戏，让我们添加下一个。

```
import datetime

class MonteCarlo(object):
    def __init__(self, board, **kwargs):
        # ...
        seconds = kwargs.get('time', 30)
        self.calculation_time = datetime.timedelta(seconds=seconds)

    # ...

    def get_play(self):
        begin = datetime.datetime.utcnow()
        while datetime.datetime.utcnow() - begin < self.calculation_time:
            self.run_simulation()
```

这里我们定义一个时间量的配置选项用于计算消耗,get_play函数将反复多次调用run_simulation函数直到时间消耗殆尽。此代码不会特别有用，因为我们没有定义run_simulation函数，所以我们现在开始写这个函数。

```
# ...
from random import choice


class MonteCarlo(object):
```

```
    def __init__(self, board, **kwargs):
        # ...
        self.max_moves = kwargs.get('max_moves', 100)

    # ...

    def run_simulation(self):
        states_copy = self.states[:]
        state = states_copy[-1]

        for t in xrange(self.max_moves):
            legal = self.board.legal_plays(states_copy)

            play = choice(legal)
            state = self.board.next_state(state, play)
            states_copy.append(state)

            winner = self.board.winner(states_copy)
            if winner:
                break
```

增加了run_simulation函数端口，无论是选择UCB1算法还是选择设置每回合遵循游戏规则的随机移动直到游戏结束。我们也推出了配置选项，以限制AI的期望移动数目。

你可能注意到我们制作self.states副本的结点，并且给它增加了新的状态。代替直接添加到self.states。这是因为self.states记录了到目前为止发生的所有游戏记录，在模拟这些探索性移动中我们不想把它做得不尽如人意。
现在，在AI运行run_simulation函数中，我们需要统计这个游戏状态。AI应该选择第一个未知游戏状态把它添加到表中。

```
class MonteCarlo(object):
    def __init__(self, board, **kwargs):
        # ...
        self.wins = {}
        self.plays = {}

    # ...
```

```
    def run_simulation(self):
        visited_states = set()
        states_copy = self.states[:]
        state = states_copy[-1]
        player = self.board.current_player(state)

        expand = True
        for t in xrange(self.max_moves):
            legal = self.board.legal_plays(states_copy)

            play = choice(legal)
            state = self.board.next_state(state, play)
            states_copy.append(state)

            # 这里的`player`以下指的是进入特定状态的玩家
            if expand and (player, state) not in self.plays:
                expand = False
                self.plays[(player, state)] = 0
                self.wins[(player, state)] = 0

            visited_states.add((player, state))

            player = self.board.current_player(state)
            winner = self.board.winner(states_copy)
            if winner:
                break

        for player, state in visited_states:
            if (player, state) not in self.plays:
                continue
            self.plays[(player, state)] += 1
            if player == winner:
                self.wins[(player, state)] += 1
```

在这里，我们添加两个字典到AI，wins和plays，其中将包含跟踪每场比赛状态的计数器。如果当前状态是第一个新状态，该run_simulation函数方法现在检测到这个调用已经被计数，而且，如果没有，增加声明palys和wins，同时初始化为零。通过设置它，这种

函数方法也增加了每场比赛的状态，最后更新wins和plays，同时在wins和plays的字典中设置那些状态。我们现在已经准备好将AI的最终

决策放在这些统计上。

```
from __future__ import division
# ...

class MonteCarlo(object):
    # ...

    def get_play(self):
        self.max_depth = 0
        state = self.states[-1]
        player = self.board.current_player(state)
        legal = self.board.legal_plays(self.states[:])

        # 如果没有真正的选择，就返回。
        if not legal:
            return
        if len(legal) == 1:
            return legal[0]

        games = 0
        begin = datetime.datetime.utcnow()
        while datetime.datetime.utcnow() - begin < self.calculation_time:
            self.run_simulation()
            games += 1

        moves_states = [(p, self.board.next_state(state, p)) for p in legal]

        # 显示函数调用的次数和消耗的时间
        print games, datetime.datetime.utcnow() - begin

        # 挑选胜率最高的移动方式
        percent_wins, move = max(
            (self.wins.get((player, S), 0) /
             self.plays.get((player, S), 1),

             p)
```

```
            for p, S in moves_states
        )

        # 显示每种可能统计信息。
        for x in sorted(
            ((100 * self.wins.get((player, S), 0) /
              self.plays.get((player, S), 1),
              self.wins.get((player, S), 0),
              self.plays.get((player, S), 0), p)
             for p, S in moves_states),
            reverse=True
        ):
            print "{3}: {0:.2f}% ({1} / {2})".format(*x)

        print "Maximum depth searched:", self.max_depth

        return move
```

我们在此步骤中添加三点。首先，我们允许，如果没有选择，或者只有一个选择，使get_play函数提前返回。其次，我们增加输出了一些调试信息，包含每回合移动的统计信息，且在淘汰赛选择阶段，将保持一种属性 ，用于根踪最大深度搜索。最后，我们增加代码用来挑选出胜率最高的可能移动，并且返回它。

但是，我们远还没有结束。目前，对于淘汰赛我们的AI使用纯随机性。对于在所有数据表中遵守游戏规则的玩家的位置我们需要UCB1算法，因此下一个尝试游戏的机器是基于这些信息。

```
# ...
from math import log, sqrt

class MonteCarlo(object):
    def __init__(self, board, **kwargs):
        # ...
        self.C = kwargs.get('C', 1.4)

    # ...


    def run_simulation(self):
```

```
    # 这里最优化的一点，我们有一个<span style="font-family: Arial, Helvetica, sans-serif;">查找</span><span style="font-family: Arial, Helvetica, sans-serif;">局部变量代替每个循环中访问一种属性</span>
    plays, wins = self.plays, self.wins

    visited_states = set()
    states_copy = self.states[:]
    state = states_copy[-1]
    player = self.board.current_player(state)

    expand = True
    for t in xrange(1, self.max_moves + 1):
        legal = self.board.legal_plays(states_copy)
        moves_states = [(p, self.board.next_state(state, p)) for p in legal]

        if all(plays.get((player, S)) for p, S in moves_states):
            # 如果我们在这里统计所有符合规则的移动，且使用它。
            log_total = log(
                sum(plays[(player, S)] for p, S in moves_states))
            value, move, state = max(
                ((wins[(player, S)] / plays[(player, S)]) +
                 self.C * sqrt(log_total / plays[(player, S)]), p, S)
                for p, S in moves_states
            )
        else:
            # 否则，只做出错误的决定
            move, state = choice(moves_states)

        states_copy.append(state)

        # 这里的`player`以下指移动到特殊状态的玩家
        if expand and (player, state) not in plays:
            expand = False
            plays[(player, state)] = 0
            wins[(player, state)] = 0
            if t > self.max_depth:
                self.max_depth = t


        visited_states.add((player, state))
```

```
            visited_states.add((player, state))

            player = self.board.current_player(state)
            winner = self.board.winner(states_copy)
            if winner:
                break

        for player, state in visited_states:
            if (player, state) not in plays:
                continue
            plays[(player, state)] += 1
            if player == winner:
                wins[(player, state)] += 1
```

这里主要增加的是检查，看看是否所有的遵守游戏规则的玩法的结果都在plays字典中。如果它们不可用，则默认为原来的随机选择。但是，如果统计信息都可用，根据该置信区间公式选择具有最高值的移动。这个公式加在一起有两部分。第一部分是这个胜率，而第二部分是一个叫做被忽略特定的缓慢增长变量名。最后，如果一个胜率差的结点长时间被忽略，那么就会开始被再次选择。这个变量名可以用配置参数c添加到__init函数上。c值越大将会触发更多可能性的探索，且较小的值会导致AI更偏向于专注于已有的较好的移动。还要注意到，当添加了一个新结点且它的深度超出self.max_depth时，从以前代码块中的the self.max_depth属性被立即更新。

这样就能生成它，如果没有错误，你现在应该有一个AI将做出合理决策的各种棋盘游戏。我留下了一个合适的模板用于读者的练习，然而我们留下了给予玩家再次使用AI玩的一种方式。这种游戏框架可以在 jbradberry/boardgame-socketserver and jbradberry/boardgame-socketplayer找到。

这是我们刚刚建立的新手玩家版本。下一步，我们将探索改善AI以供高端玩家使用。通过机器自我学习来训练一些评估函数并与结果挂钩。



Hanson1
关注 - 0
粉丝 - 3
+加关注

0 0

« 上一篇：Python时间日期函数讲解
» 下一篇：Linux quota命令参数及用法详解---Linux磁盘配额限制设置和查看命令

posted on 2017-06-30 22:40 Hanson1 阅读(60) 评论(0) 编辑 收藏

· 写给立志做码农的大学生

» 更多知识库文章...