**РОССИЙСКИЙ ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ Г.В. ПЛЕХАНОВА**

Курсовая работа

по курсу «Теория разработки современных программных комплексов»

на тему «Искусственный интеллект для настольной игры Го»

**Выполнил студент группы ДКО-131б:**

*Шемякин И.И. ДКО-13193*

**Руководитель:**

*Чапкин Н.С.*

Москва 2015

Оглавление

[Резюме 2](#_Toc440641771)

[Введение 3](#_Toc440641772)

[Игра Го 4](#_Toc440641773)

[Правила Го 4](#_Toc440641774)

[Сложности Го 5](#_Toc440641775)

[Компьютерное Го 7](#_Toc440641776)

[Эвристический алгоритм 7](#_Toc440641777)

[Игровые деревья 8](#_Toc440641778)

[Минимаксное дерево 9](#_Toc440641779)

[Метод Монте-Карло 10](#_Toc440641780)

[Метод Монте-Карло для деревьев 11](#_Toc440641781)

[UCT – Upper Confidence Bound for Trees 13](#_Toc440641782)

[Улучшения для UCT 15](#_Toc440641783)

[Распараллеливание 15](#_Toc440641784)

[Функция выкатывания 15](#_Toc440641785)

[RAVE (Rapid Action Value Estimation) 16](#_Toc440641786)

[Заключение 16](#_Toc440641787)

[Список используемой литературы 17](#_Toc440641788)

# Резюме

Древняя восточная игра Го уже давно считается огромным препятствием для технологий искусственного интеллекта. В течение многих десятилетий, компьютерное Го бросало вызов классическим методам поиска игрового дерева, который успешно реботал для шахмат и шашек. Однако, современное компьютерное Го было буквально преобразовано благодаря новой парадигме для поиска дерева на основе метода Монте-Карло для деревьев. Программы, основанные на этом методе в настоящее время играют на уровне сильных любителей, и постепенно приближаются к уровню профессиональных игроков. В своей работе я постараюсь описать и продемонстрировать наиболее современные алгоритмы Монте-Карло для игрового дерева и объяснить, как они продвинули уровень развития компьютерного Го.

# Введение

Последовательное принятие решений уже было изучено в ряде областей, начиная от теории управления, и заканчивая исследованиями в области искусственного интеллекта (ИИ). Задача последовательного принятия решений состоит в том, чтобы выбрать действия, которые добиваются некоторый долгосрочной цели (например, победы в игре), когда последствия этих действий не могут быть выявлены в течение многих последующих шагов. В этой работе я остановлюсь на новом подходе к последовательному принятию решений, который был разработан в последнее время в контексте игр для двух игроков.

Классические игры для двух игроков являются отличным тестом для ИИ. Они представляют собой закрытые микромиры с простыми правилами, которые были отобраны и совершенствовались в течение сотен или тысяч лет для того, чтобы бросить вызов игрокам-людям. Эти правила также обеспечивают четкие критерии сравнения программ и человеческого интеллекта.

Во многих играх на двух игроков, таких как шахматы, шашки и нарды, люди уже давно были превзойдены программами, сочетающими перебор (brute force) дерева игры с человеческими знаниями (эвристика).

Однако, в за последние полвека эти и другие подходы не смогли добиться схожего успеха в игре Го. Размер пространства поиска в Го является настолько большим, что оно не поддаётся грубому перебору. Кроме того, трудно охарактеризовать силу позиции или хода. По этим причинам, Го большим вызовом искусственному интеллекту. Эта область ещё ждёт триумфа, похожего по значимости на победу суперкомпьютера Deep Blue в 1997 году против Гарри Каспарова, сильнейшего на тот момент шахматиста мира.

Последние пять ознаменовались значительным прогрессом в компьютерном Го. С развитием нового метода Монте-Карло, компьютерные программы по Го добились значительных успехов, в том числе нескольких побед против профессиональных игроков. Составляющие этих программ обманчиво просты. Oни обеспечены минимальным предварительным знанием об игре Го, и вместо этого они приобретают опыт путём моделирования случайных игр самих с собой. Эти алгоритмы называются методом Монте-Карло поиска в дереве (Monte-Carlo Tree Search, сокр. **MCTS**), потому что он строит и расширяет дерево поиска, в то же время оценивая силу отдельных ходов по их успехам в ходе случайных игр.

# Игра Го

Го – древняя настольная игра для двух игроков, впервые появившаяся в Китае. На данный момент в неё играет около 27 миллионов человек по всему миру. Одной из её отличительных особенностей является простота правил, скрывающая за собой потрясающую многогранность.

Перед тем, как перейти к объяснению алгоритмов искуственного интеллекта для Го, стоит остановиться на базовых понятиях и правилах игры.

## Правила Го

Основные правила заключаются в следующем.

Чёрные и белые ходят по очереди, причём, в отличие от щахмат, в Го чёрные ходят первыми. Каждый свой ход игроки ставят по одному **камню** соответствующего цвета на доску . Стандартным размером для соревнований является , хотя и меньшие поля – в частности, 13х13 и 9х9, представленная в работе – также широко используются для обучения, и для разработки компьютерного Го.

**Группой** называется объёдинённый набор без диагоналей камней одного цвета. **Дыхание** – пустое пересечение, которое прилегает к группе камней, опять же не считая диагональных. Группа считается **захваченной**, когда у неё не остаётся дыханий. Все камни захваченной группы снимаются с поля. Правилами запрещены **суицидальные** ходы, приводящие к немедленному захвату собственной группы. Группа считается **мёртвой**, если её захват неизбежен. **Живой** же называется группа камней, захват которой невозможен.

Основной целью игры является захват **территории**, которая трактуется по-разному в разных странах, хотя в большинстве случаев все наиболее распространённые наборы правил приодят к одному результату. В Китайских правилах, используемых в данной работе, территорией считаются все пустые пересечения, которые неоспоримо лежат, внутри живых групп игрока, и сами живые камни. Каждый игрок получает по одному очку за каждое пустое пересечение, контролируемое его группами, и по одному очку за каждый камень, который остался живым к концу игры.

Белым даются бонусные очки, называемые **коми**, в качестве компенсации за то, что игру начинают чёрные. Заканчивается игра в тот момент, когда один из игроков сдаётся, или же когда оба игрока по очереди пасуют.

**Фора** – это количество камней, которые может поставить чёрный игрок перед поочерёдными ходами для компенсации силы игроков. **Равной игрой** называется игра без форы с коми, равным 6.5.

Это все базовые знания, которые необходимы для понимания игры, поэтому теперь можно продолжить разбор её цифровой составляющей.

## Сложности Го

С появлением метода Монте-Карло для деревьев в 2006 году, уровень программ для Го взлетел с уровня слабых новичков до профессионального на поле размером , и до уровня сильных любителей на поле (рис.1).

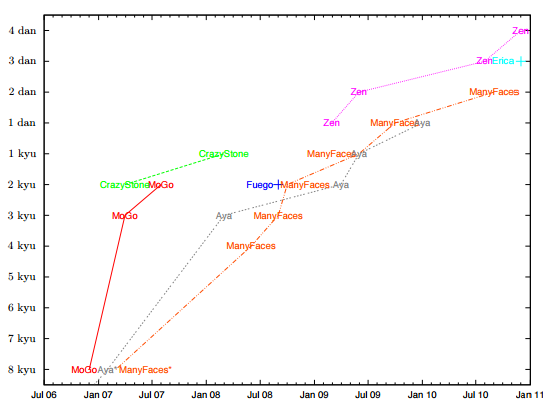


рис. 1

Го сложна для компьютерных программ по целому ряду причин. Во-первых, комбинаторная сложность игры невероятно велика.Для наглядности приведём величины, характеризующие этот вид сложности, в сравнении с шахматами. На каждом ходу в Го очень много вариантов хода: в среднем около 200, против 37 в шахматах. Длина средней игры составляет около 250 ходов, против 57 в шахматах. Общее количество возможных позиций превышает (), а количество возможных цепочек ходов насчитывает более вариантов (против [число Шеннона]) – это на сотни порядков больше, чем насчитывается атомов в видимой части вселенной!

Другим аспектом игры, представляющим сложность, является долгосрочное влияние ходов: расположение камня в начале игры может существенно повлиять на исход игры сотни ходов спустя. Базовая эвристика для оценки позиции, например расчёт материального преимущества, показала себя с лучшей стороны в шахматах и шашках. Однако, она не так полезна в Го из-за того, что преимущество одного игрока в территории зачастую компенсируется лучшей стратегической позицией оппонента. В результате, лучшая базовая эвристика оценивает позицию на уровне новичка.

Вместе, эти аспекты делают Го отличной задачей для ИИ, так как ему необходимо справляться с большим игровым пространством, долгосрочным влиянием ходов и важностью стратегической позиции одновременно. Многие проблемы последовательного принятия решений в жизни сложны из-за абсолютно таких же причин, поэтому прогресс в компьютерном Го может в итоге внести вклад в область ИИ в целом. Одним из подтверждений данного утверждения является тот факт, что метод Монте-Карло для деревьев, изначально появившийся в компьютерном Го, стал успешно применяться и в других областях ИИ.

# Компьютерное Го

Хотя все современные программы для Го используют метод Монте-Карло, в них для вторичной оценки хода используются и классические алгоритмы. Остановимся на них подробнее.

## Эвристический алгоритм

Суть эвристического алгоритма заключается в разбиении поля на небольшие сегменты и оценки полученной локальной позиции с помощью перебора или распознавания критических точек на основе паттернов (на рис.2 примеры наиболее распространённых позиций с крит. точками).

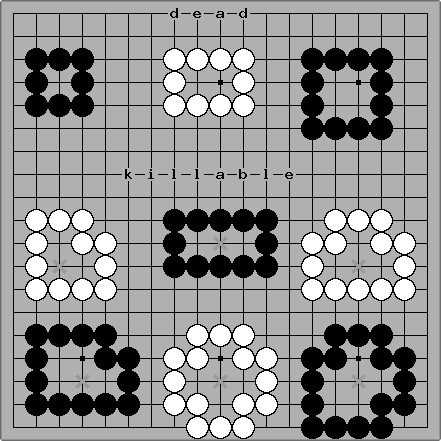


рис. 2

К преимуществам можно отнести высокую скорость и точность оценки ситуаций жизни и смерти в небольших локальных позициях (3х3, 4х4, 5х5). К недостаткам же относятся слабость в оценке стратегической позиции и неспособность учитывать в расчётах отдалённые камни.

В современных программах применяется в основном для разыгрывания стандартных начальных комбинаций (джосеки), для скоростной оценки жизни-смерти и соединения групп и смещения приоритетов поиска, например, при выходе за пределы дерева поиска. Рис. 3 представляет сравнение процента побед трёх ИИ против одного и того же ИИ (MoGo), основанного на МК, но дополненного разными дополнительными функциями: слева используется случайный поиск хода, справа – эвристический. Отсюда очевидно, что использование качественной эвристической функции значительно увеличивает общую силу ИИ.

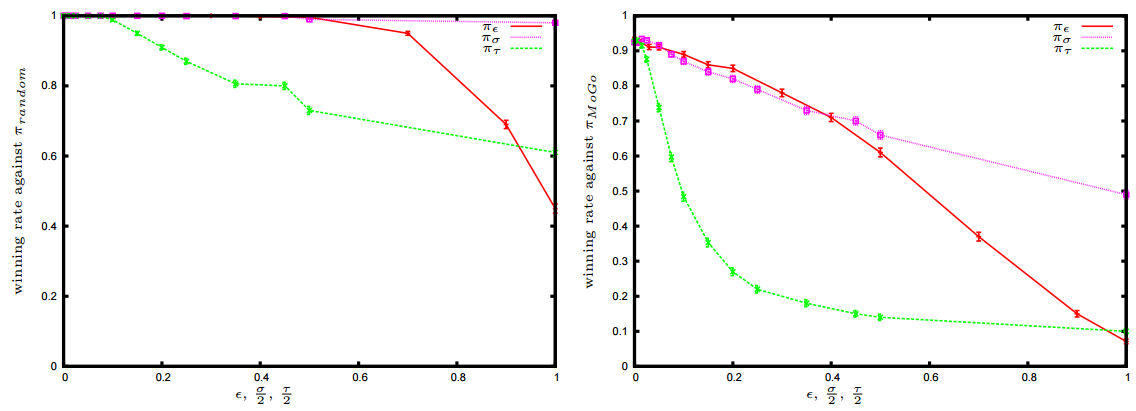


рис. 3

## Игровые деревья

Основной составляющей программ для Го, как уже было сказано, является метод Монте-Карло. Определим его область применения и базовые понятия.

Метод Монте-Карло для деревьев применяется в основном в играх для двух человек, делающих ходы по очереди. Желательными являются **детерминированность** игры и наличие **полной информации**. Детерминированность предполагает отсутствие случайных событий, таких как броски кубика или перемешивание карт. Полная информация же означает отсутствие у игроков информации, известной только им, например, «закрытых» карт на руках. Более точно, полная информация означает возможность каждого игрока посчитать разброс результатов игр (который будет нулевым, если игра детерминированная) по любой данной последовательности действий. Другой способ описания говорит о том, что оба игрока знают **состояние** игры. Не считая рокировку и правило Ко из шахмат и Го соответственно, можно сказать, что состояние игры идентично **положению** на поле, т.е. расположению на нём всех фигур.

Правила игры также определяют **конечное состояние**, в котором игра заканчивается. Каждому конечному состоянию соответствует некоторая награда для каждого из игроков. При этом в игре отсутствуют промежуточные награды, т.е. для каждого не конечного состояния награды равны нулю.

**Игровое дерево** организует все возможные будущие последовательности в структуру дерева. Корень дерева соответствует начальной позиции (и пустой последовательности действий), в то время как все остальные его части соответствуют некоторой непустой и конечной последовательности действий, каждая из которых всегда приводит к одному и тому же результату. Также важно заметить, что одна и та же позиция может ассоциировать с несколькими последовательностями действий, называемых **транспозициями**. В таком случае игру можно представить как ацикличный граф на наборе состояний.

Теперь рассмотрим игровые деревья подробнее. Так как метод Монте-Карло для деревьев достаточно сложен, начнём с его составляющей, успешно применяющейся в других играх – минимаксного дерева.

### Минимаксное дерево

Алгоритм минимаксных деревьев обычно считает конечную награду одного из игроков, например, чёрного. В этой ситуации задача белого игрока состоит в том, чтобы искать ноды дерева, в которых эта награда будет минимальной.

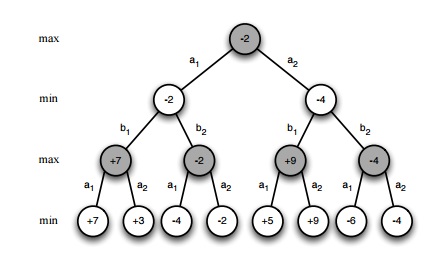


рис. 4

Тогда **оптимальное значение** игрового дерева – это значение, которое каждый из игроков может гарантировать себе при условии, что противник будет пользоваться оптимальной стратегией.

Если оптимальные значения всех потомков известны, то выбор лучшего хода родителя становится тривиальной задачей: чёрный (белый) игрок просто выбирает тот вариант хода, который максимизирует (минимизирует) награду конечного состояния.

Главный плюс минимаксного дерева состоит в том, что по нему всегда выбирается оптимальная стратегия. Минус же в том, что объём вычислений пропорционален размеру дерева, который в свою очередь экспоненциально растёт с увеличением глубины дерева.

Одним из самых используемых улучшений минимаксного дерева является альфа-бета-отсечение и итеративное углубление, что позволяет смотреть только часть дерева, и при нахождении удовлетворительного решения сразу переходить в него без оценки всего игрового пространства.

В случае отсечения важным аспектом является функция оценки позиции, которая способна с определённой точностью оценить предполагаемую награду. Программы на основе минимаксного дерева с альфа-бета-отсечением уже сильнее мировых чемпионов по шахматам, шашкам и отелло (также называется реверси).

### Метод Монте-Карло

Одной из существенных проблем использования альфа-бета отсечения в Го является запредельная сложность нахождения достаточно качественной оценочной функции для достижения удовлетворительного результата. Поэтому в качестве альтернативы конструированию оценочной функции можно формировать установку, с помощью которой приблизительно оценивать ценность состояний. **Установка** – это привязка действий к состояниям, или, другими словами, способ играть в игру. Если каждому игроку дана установка, то оценочное значение состояния можно получить с помощью **симуляции**: начать в некотором состоянии S и по очереди использовать соответствующие установки до некоторого конечного состояния E, и использовать полученную в Е награду в качестве оценки состояния S.

В некоторых играх, включая Го, проще использовать установку для формирования косвенной оценки через симуляцию, так как проще придумать относительно простую установку, приводящую к достаточно точным результатам, чем пытаться расчитать оценку напрямую. Одной из серъёзных проблем описанного подхода является высокая зависимость от выбора установки. Например, выбранная установка может выбирать оптимальное действие в 90% случаев и плохое в оставшихся 10%. Так как установка неизменчива, оценки будут страдать от систематических ошибок, так как симуляция будет приводить к одной и той же последовательности действий из данного состояния. Эти ошибки могут приводить к катастрофическим последствиям, что выльется в неточные оценки и явным дырам в стратегии, которые может использовать оппонент.

Методы Монте-Карло решают эту проблему путём явной рандомизации установки и использования **ожидаемой награды** в качестве оценки. Потенциальная польза от рандомизации двояка: она позволяет уменьшить влияние систематических ошибок и помогает отличать состояния, в которых «легко победить» от тех, в которых «сложно победить». Это различие полезно потому, что реально существующие оппоненты также несовершенны, и поэтому стоит сдвигать игру к тем состояниям, для которых есть много выигрышных стратегий.

Однако, когда установка рандомизирована, нахождение точного ожидаемого результата состояния можно оказаться такой же сложной задачей (или ещё сложнее), чем вычисление оптимального значения. К счастью, метод Монте-Карло может дать качественное приближение ожидаемой награды. Идея заключается в том, чтобы просто запустить множество симуляций с рандомизированной установкой из данного состояния, а потом брать усреднённое значение. Полученное значение называется **оценкой Монте-Карло.**

Использование метода Монте-Карло в играх было впервые задокументировано в 1973 году, когда его применили к блекджеку. Использование данного метода в играх с неполной информацией и случайными событиями вполне естественно. С другой стороны, принесение «шумов» в детерминированные игры с полной информацией уже не столь естественно. Впервые эта идея была предложена в 1990 году Абрамсоном, а реализована только в 2006-2008.

В данной работе для демонстрации метода представлен ИИ, реализованный с помощью «чистого» метода Монте-Карло.

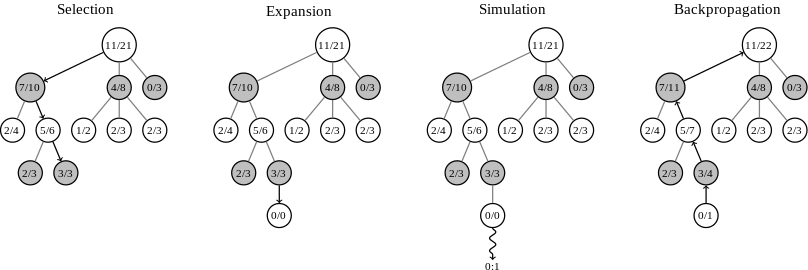
### Метод Монте-Карло для деревьев

Метод Монте-Карло для деревьев – MCTS – объединяет симуляции чистого метода Монте-Карло с просмотром игрового дерева. Во время поиска он выборочно удлиняет части игрового дерева. Так же, как в минимаксном поиски, каждый нод дерева соответствует одному состоянию игры. Но, в отличие от минимаксного поиска, значение нодов получается с помощью оценки Монте-Карло.

В объяснении чистого метода Монте-Карло предполагалось, что в симуляциях используется одна неизменная установка. Одна из ключевых идей MCTS состоит в постепенной адаптации и улучшении этой установки.

С увеличением количества проведённых симуляций растёт игровое дерево, и оценки Монте-Карло в нодах становятся всё точнее, давая всё больше информации, позволяющей смещать поиск в сторону дочерних нодов с высокими значениями награды. В среднем, это смещение улучшает установку, приближая результаты симуляций к оптимальным. Чем сильнее смещение, тем более выборочным будет игровое дерево, что выливается в неравномерное дерево, которое более глубоко расширяется в сторону нодов с высокими значениями. В любом случае, вычисленное дерево будет содержать лишь малую часть полного возможного дерева игры. В какой-то момент симуляция придёт в состояние, которое не отражено в вычисленном дереве. В такие моменты алгоритм должен вернуться к единой фиксированной установке, и начать адаптацию установки заново. Этот процесс называется **выкатыванием**.

В целом, весь MCTS состоит из данного цикла: пока не выполнится критерий остановки (обычно это время вычислений или количество симуляций), метод крутится по 4 фазам: спуск по дереву, выкатывание, обновление значений, рост дерева.



В фазе спуска MCTS итеративно выбирает дочерний нод с наибольшей **выгодой,** которая будет более подробно рассмотрена в следующей секции. В конце фазы спуска, т.е. когда достигнут лепесток дерева, случайная установка используется для симуляции игры, пока не будет достигнуто конечное состояние. В конце фазы выкатывания, финальная награда записывается для определения награды для игрока. В фазе обновления полученная статистика (количество побед и заходов) для каждого нода обновляется в соответствии с результатами симуляций. В фазе роста первое состояние, полученное в выкатывании, добавляется в дерево, и инициализируется его статистика.

### UCT – Upper Confidence Bound for Trees

Крайне желательной характеристикой любого поиска по игровому дереву является последовательность, т.е. при наличии достаточного количества времени, поисковый алгоритм должен в итоге находить все оптимальные значения для всех нодов дерева, и должен быть способен выбирать оптимальное действие для корневого состояния. Алгоритм UCT является последовательной версией MCTS.

Если бы все оценки листьев были оптимальными значениями, то можно было бы достичь последовательности в родительских нодах при помощи жадного выбора, который просто выбирает действие с наивысшей наградой в каждом ноде. Если все дочерние ноды хранят оптимальные значения, то жадный выбор в дальнейшем получает набор оптимальных действий. При помощи индукции, оценки всех нодов рано или поздно станут оптимальными, и в итоге эта процедура выберет оптимальное действие в корне.

Однако, оценки, как правило, неоптимальны по двум причинам:

1. Установка рандомизирована, так что в значениях есть некоторый врождённый элемент случайности.
2. Установка несовершенна.

Поэтому выбор действия с наивысшей оценкой может привести к неоптимальной игре, например если оценка оптимального действия была с самого начала недооценена. Поэтому хотя бы иногда необходимо выбирать действия, являющиеся неоптимальными согласно текущим оценкам.

Задача того, когда и как выбирать оптимальный или неоптимальный вариант была глубоко изучена в простейшей случайной задаче выбора, получившей название «задачи многорукого бандита». В этой задаче игра заканчивается после первого же действия, и получаемая игроком случайная награда зависит только от выбранного действия. Суть задачи в том, чтобы максимизировать суммарную выгоду игрока, т.е. быстро найти действие с наивысшей ожидаемой наградой без чрезмерной потери при поиске. Одна простая, но эффективная стратегия заключается в том, чтобы всегда выбирать действие с наивысшей оценкой с оптимистичным предположением о неопределённости оценок. Таким образом, каждое действие либо получается оптимальным, либо уменьшает связанную с ним неопределённость, поэтому неоптимальные действия не могут выбираться бесконечно.

Стратегия выбора действия с наивысшей оптимистичной оценкой получила название «закона об оптимизме при встрече с неопределённостью».

Одной из относительно несложных реализаций является вычислений верхней границы доверия для каждой оценки с помощью неравенства Хёфдинга. Применение этой идеи к MCTS и даёт в результате алгоритм UCT, где оценка каждого нода получается по формуле:

Где С – подстраивающая константа, – количество симуляций, в которых ход был выбран в состоянии , – суммарная награда, собранная из конечных состояних по итогам симуляций, – количество симуляций из состояния , – бонус исследования. Все оценки смещаются на величину бонуса, который является наибольшим для тех нодов, которые посещались редко, и поэтому являются наиболее неопределёнными.

ИИ на базе UCT с параллельными вычислениями также реализован в данной работе.

# Улучшения для UCT

## Распараллеливание

Классические алгоритмы поиска по дереву игры, такие как альфа-бета-отсечение, представляют значительные трудности в распараллеливании. К счастью, в MCTS намного проще реализовать параллельные вычисления: так, MoGo распределяли на 800 процессоров датского суперкомпьютера «Гюйгенс».

Тем не менее, стоит проявлять осторожность при реализации параллелизма в MCTS. Зачастую параллельный MCTS проявляет себя хуже, чем однопоточный, при одинаковом числе симуляций. Причина в том, что однопоточный MCTS увеличивает дерево выборочно, опираясь на результаты предыдущих симуляций, в то время как параллельной реализации приходится проводить больше слепых симуляций, не учитывающих результаты одновременных вычислений. В худшем случае параллельности, игровое дерево не строится, и MCTS ведёт себя точно так же, как «наивный» алгоритм Монте-Карло.

## Функция выкатывания

Сила ИИ на базе MCTS в значительной степени зависит от рандомизированной установки во время фазы выкатывания. Тщательно подобранная функция для выкатывания может существенно усилить такой ИИ по сравнению со случайной функцией выбора. Но если функция станет слишком детерминированной, то алгоритм становится очень чувствителен к ошибкам, которые создаёт установка выкатывания. Поэтому усиление установки может закончится ослаблением ИИ в целом. Более того, более сложная установка может дольше выполняться, что ухудшит силу ИИ в целом, даже если он будет показывать лучшие результаты за одну симуляцию.

Пока что существует три подхода к этой проблеме:

1. Первый подход реализован в MoGo. Там используется сделанная вручную функция выкатывания, которая выбирает случайную позицию из набора рекомендованных экспертных паттернов.
2. Второй подход аналогичен, но использует обучение под наблюдением для создания весов паттернов, чтобы веса соответствовали выбору экспертов-наблюдателей.
3. Третий подход заключается в так называемой балансироке симуляций, и пытается найти случайную установку, для которой оценки будут близки к оптимальным. Производится с помощью обучения через глубокий поиск, в котором, в отличие от предыдущего подхода, не штрафуются расхождения, так как уменьшение случайности функции может ослабить ИИ в целом.

## RAVE (Rapid Action Value Estimation)

Данный алгоритм реализует эвристику под общим названием all-moves-as-first (AMAF), считающую, что ходы в других частях поля не влияют на некоторую выбранную локальную позицию, и распространяет оценку некоторого действия A на всё поддерево состояния, в котором оно делается, считая, что это действие произведёт примерно похожий эффект на весь сегмент дерева.

Алгоритм придаёт ощутимое ускорение вычислениям, но уменьшает точность по сравнению с «чистым» MCTS/UCT.

# Заключение

Данная работа представила ключевые компонента метода Монте-Карло для деревьев и несколько важных расширений, которые позволили программам по Го добиться своих текущих характеристик. Эти алгоритмы привели к существенному увеличению силы ИИ для игры, впервые позволив им соперничать с сильными оппонентами-людьми.

Го воплощает в себе преграды, встречаемые в во многих проблемах последовательного принятия решений в реальном мире. У неё огромное пространство поиска, очень высокий фактор ветвления, отложенные последствия действий, и функция оценивания которую тяжело построить или выучить с опытом. Метод Монте-Карло для деревьев представляет новую парадигму для планирования в этой сложной сфере, которая, возможно, сможет использоваться далеко за пределами игр, для которых была разработана изначально.

# Список используемой литературы

1. Special issue on Monte Carlo techniques and computer Go. In C.-S. Lee, M. Muller, and O. Teytaud, editors, ¨ IEEE Transactions on Computational Intelligence and AI in Games, volume 2. 2010.
2. B. Abramson. Expected-outcome: a general model of static evaluation. IEEE Transactions on Pattern Analysis and Machine Intelligence, 12:182–193, 1990
3. A. Bourki, G. Chaslot, M. Coulm, V. Danjean, H. Doghmen, J.-B. Hoock, T. Herault, A. Rimmel, F. Teytaud, O. Teytaud, P. Vayssiere, and Z. Yu. Scalability and Parallelization of Monte-Carlo Tree Search. In Proc. Intl. Conf. on Computers and Games, 2010.
4. B. Bouzy and B. Helmstetter. Monte-Carlo Go developments. In H. I. E.A. Heinz H.J. van den Herik, editor, 10th Advances in Computer Games, pages 159–174, Graz, 2003. Kluwer Academic Publishers.
5. S. Bubeck, R. Munos, G. Stoltz, and C. Szepesv´ari. Online optimization in X-armed bandits. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, NIPS-21, pages 201–208. MIT Press, 2009.
6. P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite time analysis of the multiarmed bandit problem. Machine Learning,

# Листинг

## Program.cs

using System;

using System.IO;

using System.Runtime.InteropServices;

using System.Text;

namespace dotNetGo

{

public class Program

{

private const string alphabet = "abcdefghijklmnopqrstuvwxyz"; //for recording

private static StringBuilder gameRecord = new StringBuilder();

private static bool \_isHumanChosen;

public static void Main(string[] args)

{

IPlayer black = ChoosePlayer(1);

if (black == null)

{

Console.WriteLine("User decided to exit");

return;

}

IPlayer white = ChoosePlayer(2);

if (white == null)

{

Console.WriteLine("User decided to exit");

return;

}

PlayGame(black, white);

}

private static IPlayer ChoosePlayer(int playerNumber)

{

if (playerNumber != 1 && playerNumber != 2)

throw new ArgumentOutOfRangeException("playerNumber");

string playerColor = playerNumber == 1 ? "black" : "white";

Console.WriteLine("Choose {0} player", playerColor);

Console.WriteLine("1. UCT");

Console.WriteLine("2. MonteCarlo");

Console.WriteLine("3. MonteCarlo Random");

if (\_isHumanChosen == false)

Console.WriteLine("4. Human");

Console.WriteLine("Anything else: Exit program");

Console.Write("-> ");

string choiceString = Console.ReadLine();

int choice;

if (String.IsNullOrWhiteSpace(choiceString) == true || int.TryParse(choiceString, out choice) == false)

return null;

switch (choice)

{

case 1:

Console.WriteLine("Choose UCT simulation mode:");

Console.WriteLine("1. Random");

Console.WriteLine("2. Smart");

Console.WriteLine("Anything else: Exit program");

int simulationMode = 0;

if (int.TryParse(Console.ReadLine(), out simulationMode) == false || simulationMode < 1 || simulationMode > 2)

return null;

switch (simulationMode)

{

case 1:

return new MonteCarloUCT((byte)playerNumber, true, false);

case 2:

return new MonteCarloUCT((byte)playerNumber, false, false);

default:

return null;

}

case 2:

return new MonteCarlo();

case 3:

return new MonteCarloRandom();

case 4:

if (\_isHumanChosen == true)

return null;

\_isHumanChosen = true;

return new HumanPlayer();

default:

return null;

}

}

public static void PlayGame(IPlayer blackPlayer, IPlayer whitePlayer)

{

gameRecord.AppendLine("(;FF[4]GM[1]SZ[9]AP[dotNetGo]");

gameRecord.AppendLine(String.Format("PB[{0}]", blackPlayer.Name));

gameRecord.AppendLine("HA[0]");

gameRecord.AppendLine(String.Format("PW[{0}]", whitePlayer.Name));

gameRecord.AppendLine("KM[6.5]");

gameRecord.AppendLine("RU[Chinese]");

gameRecord.AppendLine("");

gameRecord.AppendLine("");

Board board = new Board();

while (board.IsGameOver() == false)

{

Move move;

switch (board.ActivePlayer)

{

case 1:

move = blackPlayer.GetMove();

break;

default: //case 2:

move = whitePlayer.GetMove();

break;

}

if (blackPlayer.ReceiveTurn(move) == false)

throw new ImpossibleException("somehow invalid turn made it through", "PlayGame");

if (whitePlayer.ReceiveTurn(move) == false)

throw new ImpossibleException("somehow invalid turn made it through", "PlayGame");

if (move.row >= 0 && move.column >= 0)

gameRecord.AppendFormat(";{0}[{1}{2}]", board.ActivePlayer == 1? "B": "W", alphabet[move.column], alphabet[move.row]);

if (board.PlaceStone(move) == false)

throw new ImpossibleException("somehow invalid turn made it through", "PlayGame");

Console.WriteLine(board);

//Console.ReadLine();

}

switch (board.State)

{

case Board.GameState.BlackSurrendered:

Console.WriteLine("White won by resignation, last position:");

break;

case Board.GameState.WhiteSurrendered:

Console.WriteLine("Black won by resignation, last position:");

break;

case Board.GameState.DoublePass:

double blackScore, whiteScore;

board.DetermineWinner(out blackScore, out whiteScore);

gameRecord.AppendFormat(";RE[{0}+{1}]", blackScore > whiteScore?"B":"W", Math.Abs(blackScore-whiteScore));

Console.WriteLine(board);

Console.WriteLine("Turn: {0}", board.TurnNumber);

Console.WriteLine("Black score: {0}; White score: {1}", blackScore, whiteScore);

Console.WriteLine("last position:");

break;

}

Console.WriteLine(board);

gameRecord.Append(")");

DateTime dt = DateTime.Now;

string filename = String.Format("{0}-{1}-{2}-{3}-{4}-{5}.sgf",

dt.Year, dt.Month, dt.Day, dt.Hour, dt.Minute, dt.Second);

File.WriteAllText(filename, gameRecord.ToString(), Encoding.UTF8);

}

}

}

## Board.cs

using System;

using System.Collections.Generic;

using System.Net.Configuration;

using System.Runtime.CompilerServices;

using System.Text;

using System.Xml.XPath;

namespace dotNetGo

{

class Board

{

public enum GameState

{

GameIsNotOver,

BlackSurrendered,

WhiteSurrendered,

DoublePass

}

//board size

public const int Size = GameParameters.BoardSize;

private const double \_komi = GameParameters.Komi;

//cardinal directions

private static readonly int[,] CardinalDirections = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};

private static readonly int[,] DiagonalDirections = {{-1, 1}, {1, 1}, {1, -1}, {-1, -1}};

public const int DirectionCount = 4;

private int \_passes = 0;

//turn number - needed for simulations

public int TurnNumber { get; private set; }

public int Passes

{

get

{

return \_passes;

}

private set

{

\_passes = value;

if (\_passes >= 2)

State = GameState.DoublePass;

}

}

public byte ActivePlayer { get; private set; }

public GameState State { get; private set; }

public byte OppositePlayer

{

get { return (byte)(3 - ActivePlayer); }

}

private readonly bool[,] \_visited = new bool[Size,Size];

private readonly byte[,] \_buffer = new byte[Size,Size];

//needed for ko checks

private readonly byte[,] \_lastPosition = new byte[Size,Size];

readonly byte[,] \_board = new byte[Size, Size];

public byte this[int i, int j]

{

get { return \_board[i, j]; }

set { \_board[i, j] = value; }

}

public byte this[Move m]

{

get { return \_board[m.row, m.column]; }

set { \_board[m.row, m.column] = value; }

}

public byte[,] GetBoard()

{

return \_board;

}

public Board()

{

ActivePlayer = 1;

TurnNumber = 1;

}

public void CopyStateFrom(Board b)

{

TurnNumber = b.TurnNumber;

Passes = b.Passes;

ActivePlayer = b.ActivePlayer;

State = b.State;

for (int i = 0; i < Size; i++)

for (int j = 0; j < Size; j++)

{

\_visited[i, j] = b.\_visited[i, j];

\_board[i, j] = b[i, j];

\_lastPosition[i, j] = b.\_lastPosition[i, j];

}

}

public bool PlaceStone(Move move)

{

if (State != GameState.GameIsNotOver)

return false;

if (move.row == -1 && move.column == -1)

{

Pass();

return true;

}

//check if the move is on the board

if (IsOnBoard(move) == false)

{

return false;

}

//check if the intersection is already occupied

if (\_board[move.row, move.column] != 0)

{

return false;

}

//check if the move is forbidden because of the Ko rule

if (IsKo(move.row, move.column))

{

return false;

}

Array.Copy(\_board, \_buffer, \_board.Length);

this[move] = ActivePlayer;

//if there is an enemy dragon nearby, it won't contain newly-placed stone - have to check each one individually

Array.Clear(\_visited, 0, \_visited.Length);

bool isSuicide = true; //для более быстрого определения возможности хода. Если рядом с новым камнем есть свободное пересечение, то это точно не суицид

for (int i = 0; i < DirectionCount; i++) //first check opponent's dragons

{

int testRow = move.row + CardinalDirections[i, 0];

int testCol = move.column + CardinalDirections[i, 1];

//если клетка находится за доской или на ней стоит союзный камень - пропускаем. проверка союзных камней будет позже

//if an intersection is outside the board or has allied stone - skip it for now. allied checks will come later

if (IsOnBoard(testRow, testCol) == false || \_board[testRow, testCol] == ActivePlayer)

continue;

else if (\_board[testRow, testCol] == 0)

//if a neighbouring intersection is empty, then the new stone will definitely have at least 1 liberty

{

isSuicide = false;

continue;

}

if (\_board[testRow, testCol] == OppositePlayer)

{

if (\_visited[testRow, testCol] == true)

continue;

else

{

Array.Clear(\_visited, 0, \_visited.Length);

if (IsGroupDead(testRow, testCol) == true)

{

RemoveDragon(testRow, testCol);

}

}

}

}

Array.Clear(\_visited, 0, \_visited.Length);

//если рядом с новым камнем есть дракон, то он включает в себя и этот камень

//if there is a nearby friendly dragon, it will contain newly-placed stone

if (isSuicide == true && IsGroupDead(move.row, move.column) == true)

{

this[move] = 0;

return false;

}

ActivePlayer = OppositePlayer;

//TODO: possibly add ko checks

Passes = 0;

TurnNumber++;

Array.Copy(\_buffer, \_lastPosition, \_lastPosition.Length);

return true;

}

int RemoveDragon(int row, int col)

{

if (IsOnBoard(row, col) == false || IsFree(row, col) == true)

return 0;

if (\_board[row, col] == ActivePlayer)

//if we encounter an active's player stone - skip it, because suicide is forbidden

return 0;

//AT THIS POINT, we know that an intersection is on board and it contains an opponent's stone

int result = 1;

\_board[row, col] = 0;

for (int i = 0; i < DirectionCount; i++)

{

int testRow = row + CardinalDirections[i, 0];

int testCol = col + CardinalDirections[i, 1];

result += RemoveDragon(testRow, testCol);

}

return result;

}

int RemoveDragon(Move m)

{

return RemoveDragon(m.row, m.column);

}

public void Pass()

{

ActivePlayer = OppositePlayer;

TurnNumber++;

Passes++;

}

//counts liberties of a stone group that include the stone at coordinates of m

//return values:

//-1 if m is empty space, or if this space has already been visited (to remove redundant dragon checks)

//number of liberties of the dragon otherwise

private bool IsGroupDead(int row, int col)

{

if (\_board[row, col] == 0 || \_visited[row, col] == true)

return false;

\_visited[row, col] = true;

for (int i = 0; i < DirectionCount; i++)

{

int testRow = row + CardinalDirections[i, 0];

int testCol = col + CardinalDirections[i, 1];

if (IsOnBoard(testRow, testCol) == false || \_visited[testRow, testCol] == true)

continue;

if (IsFree(testRow, testCol))

{

return false;

}

else if (\_board[testRow, testCol] == \_board[row, col])

if (IsGroupDead(testRow, testCol) == false)

return false;

}

return true;

}

public bool IsFree(int row, int col)

{

return IsOnBoard(row, col) && \_board[row, col] == 0;

}

public bool IsFree(Move m)

{

return IsOnBoard(m) && \_board[m.row, m.column] == 0;

}

public bool IsOnBoard(int row, int col)

{

return row >= 0 & row < Size && col >= 0 && col < Size;

}

public bool IsOnBoard(Move m)

{

return m.row >= 0 & m.row < Size && m.column >= 0 && m.column < Size;

}

public int DetermineWinner(out double blackScore, out double whiteScore)

{

whiteScore = 0;

blackScore = 0;

if (State == GameState.BlackSurrendered)

return 2;

if (State == GameState.WhiteSurrendered)

return 1;

if (IsGameOver() == false)

return 0;

for (int i = 0; i < Size; i++)

{

for (int j = 0; j < Size; j++)

{

switch (\_board[i, j])

{

case 1:

blackScore++;

break;

case 2:

whiteScore++;

break;

case 0:

switch (IsEye(i, j))

{

case 1:

blackScore++;

break;

case 2:

whiteScore++;

break;

default:

throw new Exception("Scoring error");

}

break;

}

}

}

whiteScore += \_komi;

if (blackScore > whiteScore)

return 1;

else return 2;

}

public int DetermineWinner()

{

int whiteScore = 0;

int blackScore = 0;

if (State == GameState.BlackSurrendered)

return 2;

if (State == GameState.WhiteSurrendered)

return 1;

if (IsGameOver() == false)

return 0;

for (int i = 0; i < Size; i++)

{

for (int j = 0; j < Size; j++)

{

switch (\_board[i, j])

{

case 1:

blackScore++;

break;

case 2:

whiteScore++;

break;

case 0:

switch (IsEye(i, j))

{

case 1:

blackScore++;

break;

case 2:

whiteScore++;

break;

default:

break;

}

break;

}

}

}

whiteScore += (int)\_komi;

if (blackScore <= whiteScore)

return 2;

return 1;

}

public void Surrender()

{

switch (ActivePlayer)

{

case 1:

State = GameState.BlackSurrendered;

break;

case 2:

State = GameState.WhiteSurrendered;

break;

}

}

//checks whether the game is over

//returns true if there all empty spaces are 1-space eyes

//returns false when there are potential moves left

public bool IsGameOver()

{

if (State != GameState.GameIsNotOver)

return true;

//now check if all existing empty intersection are eyes. If they are - the game is definitely over

for (int i = 0; i < Size; i++)

{

for (int j = 0; j < Size; j++)

{

if (IsEye(i, j) == 0)

return false;

}

}

return true;

}

public override string ToString()

{

StringBuilder sb = new StringBuilder(" ");

for (int i = 0; i < Size; i++)

sb.Append((i%10).ToString());

sb.AppendLine();

for (int i = 0; i < Size; i++)

{

sb.Append(String.Format("{0} ", i%10));

for (int j = 0; j < Size; j++)

{

switch (\_board[i, j])

{

case 1:

sb.Append("b");

break;

case 2:

sb.Append("w");

break;

default:

sb.Append(".");

break;

}

}

sb.AppendLine();

}

// Program.ToStringSpan += DateTime.Now - start;

return sb.ToString();

}

public Board Clone()

{

Board result = new Board();

result.CopyStateFrom(this);

return result;

}

//return values:

//0 - not an eye

//1 - black eye

//2 - white eye

//an intersection is an eye when all immediately surrounding stones belong to the same dragon - also fixes false eyes

public int IsEye(int row, int col)

{

if (IsOnBoard(row, col) == false || IsFree(row, col) == false)

{

return 0;

}

int black = 0;

int white = 0;

for (int i = row - 1; i <= row + 1; i++)

{

for (int j = col - 1; j <= col + 1; j++)

{

if (i < 0 || i >= Size || j < 0 || j >= Size || i == j)

{

black++;

white++;

continue;

}

switch (\_board[i,j])

{

case 1:

black++;

break;

case 2:

white++;

break;

}

}

}

if (row == 0 || row == Size-1)

{

return black == 8 ? 1 : white == 8 ? 2 : 0;

}

else if (col == 0 || col == Size - 1)

{

return black == 8 ? 1 : white == 8 ? 2 : 0;

}

else

return black >= 7 ? 1 : white >= 7 ? 2 : 0;

}

public int IsEye(Move move) //false eyes fixed

{

return IsEye(move.row, move.column);

}

public bool IsKo(int row, int col)

{

int differences = 0;

if (\_lastPosition[row, col] != ActivePlayer)

return false;

for (int i = 0; i < Size; i++)

{

for (int j = 0; j < Size; j++)

{

if (\_board[i, j] != \_lastPosition[i, j])

differences++;

}

if (differences > 2)

return false;

}

//AT THIS POINT: we know that one of the differences is the current point, and there are 2 of them in total

//the other one MUST be adjacent (cannot be otherwise), so it is Ko and the move is forbidden

return true;

}

}

}

## HumanPlayer.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace dotNetGo

{

class HumanPlayer : IPlayer

{

Board b = new Board();

public Move GetMove()

{

Board cloneBoard = b.Clone();

while (true)

{

Console.WriteLine("Enter row and column for move coordinates, split by space");

string moveCoordsString = Console.ReadLine();

if (String.IsNullOrWhiteSpace(moveCoordsString) == true)

continue;

string[] coords = moveCoordsString.Split(' ');

if (coords.Length != 2)

continue;

int row;

int column;

if (int.TryParse(coords[0], out row) == true)

{

if (int.TryParse(coords[1], out column) == true)

{

Move result = new Move(row, column);

if (cloneBoard.PlaceStone(result) == true)

return result;

}

}

}

}

public bool ReceiveTurn(Move m)

{

return b.PlaceStone(m);

}

public string Name

{

get { return "Human"; }

}

}

}

## UCTNode.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace dotNetGo

{

class UCTNode

{

public static double UCTK = GameParameters.UCTK;

public Board BoardState { get; set; }

public UCTNode Parent { get; set; }

public List<UCTNode> Children { get; set; }

public Move Position { get; set; } // position of move

public int Wins { get; private set; }

public int Visits { get; private set; }

public bool IsSolved { get; set; }

public double SolvedScore { get; set; }

public int SolvedWinner { get; set; }

public bool HasChildren { get; set; }

public override bool Equals(object obj)

{

UCTNode un = obj as UCTNode;

if (un == null)

return false;

return Position.row == un.Position.row && Position.column == un.Position.column;

}

public override int GetHashCode()

{

return Position.row\*GameParameters.BoardSize + Position.column;

}

public double GetUctValue()

{

if (IsSolved)

return -1;

return Visits > 0 ? Winrate + UCTK\*Math.Sqrt(Math.Log(Parent.Visits)/Visits) : 11111;

}

public double Winrate

{

get

{

if (Visits > 0)

return (double) Wins/Visits;

return -1;

}

}

public UCTNode(UCTNode parent, Move m, Board boardState)

{

if (m == null || boardState == null)

throw new ArgumentNullException("m");

BoardState = boardState.Clone();

Parent = parent;

Children = null;

Position = new Move(m);

Wins = 0;

Visits = 0;

}

public void CreateChildren()

{

lock (this)

{

int size = Board.Size;

Board b = BoardState;

if (Children != null)

return;

if (Parent == null || Parent.Children == null)

{

Children = new List<UCTNode>(size\*size);

}

else

{

Children = new List<UCTNode>(Parent.Children.Count);

}

for (int i = 0; i < size; i++)

{

for (int j = 0; j < size; j++)

{

//is on empty space on the board

if (b[i, j] == 0 && b.IsEye(i, j) != b.ActivePlayer)

{

Board anotherCloneBoard = b.Clone();

Move m = new Move(i, j);

if (anotherCloneBoard.PlaceStone(m) == true)

Children.Add(new UCTNode(this, m, anotherCloneBoard));

}

}

}

Children.Shuffle();

HasChildren = true;

}

}

public void Update(int wins)

{

if (wins < 0)

throw new ArgumentOutOfRangeException("wins");

Visits++;

Wins += wins;

}

public UInt64 MeasureTree()

{

UInt64 result = 1;

if (Children == null) return result;

foreach (UCTNode child in Children)

{

result += child.MeasureTree();

}

return result;

}

public UInt64 CountSolvedNodes()

{

UInt64 result;

result = IsSolved ? 1ul : 0ul;

if (Children == null) return result;

foreach (UCTNode child in Children)

{

result += child.CountSolvedNodes();

}

return result;

}

public override string ToString()

{

return String.Format("{0}.{1}; {2}: {3}/{4} : {5:F5}", Position.row, Position.column, Winrate, Wins, Visits, GetUctValue());

}

}

}

## RandomGen.cs

using System;

namespace dotNetGo

{

public static class RandomGen

{

private static Random \_global = new Random();

[ThreadStatic]

private static Random \_local;

public static int Next()

{

Random inst = \_local;

if (inst == null)

{

int seed;

lock (\_global) seed = \_global.Next();

\_local = inst = new Random(seed);

}

return inst.Next();

}

public static int Next(int max)

{

Random inst = \_local;

if (inst == null)

{

int seed;

lock (\_global) seed = \_global.Next();

\_local = inst = new Random(seed);

}

return inst.Next(max);

}

public static int Next(int min, int max)

{

Random inst = \_local;

if (inst == null)

{

int seed;

lock (\_global) seed = \_global.Next();

\_local = inst = new Random(seed);

}

return inst.Next(min, max);

}

}

}

## MonteCarloUCT.cs

using System;

using System.Linq;

using System.Threading;

using System.Threading.Tasks;

namespace dotNetGo

{

internal class MonteCarloUCT : IPlayer

{

private const int Size = GameParameters.BoardSize;

public UCTNode Root;

private byte \_player { get; set; }

private readonly int \_sims;

private int \_doneSims;

private bool \_randomUCT;

private bool \_resetTreeEachTurn;

[ThreadStatic]

private static Board \_boardClone;

[ThreadStatic]

private static Move[] \_availableMoves;

public MonteCarloUCT(byte player, bool randomSims, bool resetTreeEachTurn)

{

if (player != 1 && player != 2)

throw new ArgumentOutOfRangeException("player");

Root = new UCTNode(null, new Move(-5, -5), new Board());

Root.CreateChildren();

\_player = player;

\_randomUCT = randomSims;

\_resetTreeEachTurn = resetTreeEachTurn;

\_sims = GameParameters.UCTSimulations;

}

public UCTNode GetBestChild(UCTNode root)

{

UCTNode bestChild = null;

int bestVisits = -1;

foreach (UCTNode child in Root.Children)

{

if (child.IsSolved && child.SolvedWinner == \_player)

{

return child;

}

if (child.Visits > bestVisits)

{

bestChild = child;

bestVisits = child.Visits;

}

}

return bestChild;

}

public const double UCTK = GameParameters.UCTSimulations; // theoretically should be 0.44 = sqrt(1/5)

// Larger values give uniform search

// Smaller values give very selective search

public UCTNode UCTSelect(UCTNode node)

{

UCTNode result = null;

double bestUCT = 0;

foreach (UCTNode child in node.Children)

{

if (child.IsSolved == true)

continue;

// double uctvalue = child.Visits > 0 ? child.GetUctValue() : 111111;

double uctvalue = child.GetUctValue();

if (uctvalue > bestUCT)

{

bestUCT = uctvalue;

result = child;

}

}

return result;

}

private int PlayRandomGame(UCTNode node)

{

\_boardClone.CopyStateFrom(node.BoardState);

int turnsSimulated = 0;

while (turnsSimulated < GameParameters.GameDepth && \_boardClone.IsGameOver() == false)

{

turnsSimulated++;

Move m = new Move(-5, -5);

do

{

m.row = RandomGen.Next(-1, GameParameters.BoardSize);

m.column = RandomGen.Next(-1, GameParameters.BoardSize);

} while (\_boardClone.PlaceStone(m) == false);

}

int winner = \_boardClone.DetermineWinner();

return winner;

}

private int GetAvailableMoves(Board b)

{

if (\_availableMoves == null)

\_availableMoves = new Move[Size \* Size+1];

int moveCount = 0;

for (int i = 0; i < Size; i++)

{

for (int j = 0; j < Size; j++)

{

//is on empty space on the board and not a friendly eye

if (b[i, j] == 0 && b.IsEye(i, j) != b.ActivePlayer)

{

\_availableMoves[moveCount++] = new Move(i, j);

}

}

}

return moveCount;

}

private int PlayLessRandomGame(UCTNode node)

{

if (\_availableMoves == null)

\_availableMoves = new Move[Size\*Size+1];

\_boardClone.CopyStateFrom(node.BoardState);

int turnsSimulated = 0;

while (turnsSimulated < GameParameters.GameDepth && \_boardClone.IsGameOver() == false)

{

turnsSimulated++;

int moveCount = GetAvailableMoves(\_boardClone);

\_availableMoves.Shuffle(moveCount);

Move pass = new Move(-1, -1); //add pass to possible moves

\_availableMoves[moveCount++] = pass;

for (int i = 0; i < moveCount; i++)

{

if (\_boardClone.PlaceStone(\_availableMoves[i]) == true)

{

break;

}

}

}

int winner = \_boardClone.DetermineWinner();

return winner;

}

public bool ReceiveTurn(Move m)

{

if (Root.Children != null)

{

foreach (UCTNode child in Root.Children)

{

if (child.Position.Equals(m))

{

Console.WriteLine("UCTTurbo-{0} had {1} nodes, lost {2} nodes and now has {3} nodes", \_player==1?"Black":"White", Root.MeasureTree(), Root.MeasureTree()-child.MeasureTree(), child.MeasureTree());

Root = child;

Root.Parent.Children = null;

child.Parent = null;

if (child.Children == null)

child.CreateChildren();

return true;

}

}

}

Board newBoard = Root.BoardState.Clone();

if (newBoard.PlaceStone(m) == false)

throw new ArgumentException("invalid turn");

Console.WriteLine("UCTTurbo-{0} had {1} nodes, lost {1} nodes and now has {2} nodes", \_player == 1 ? "Black" : "White", Root.MeasureTree(), 1);

Root.Children = null; //break the link for garbage collection

UCTNode newRoot = new UCTNode(null, new Move(m), newBoard);

newRoot.CreateChildren();

Root = newRoot;

return true;

}

public string Name

{

get { return "UCT"; }

}

private int PlaySimulation(UCTNode n)

{

if (\_boardClone == null)

\_boardClone = new Board();

int randomWinner = 0;

if (n.IsSolved == true) //should always be false (only for single thread! - can be true for multiple threads)

{

int solvedCurrentPlayerWins = n.SolvedWinner == \_player ? 1 : 0;

n.Update(solvedCurrentPlayerWins); //update node (Node-wins are associated with moves in the Nodes)

return n.SolvedWinner;

}

if (n.Children == null && n.Visits < GameParameters.UCTExpansion && n.IsSolved == false)

{

if (\_boardClone == null)

\_boardClone = new Board();

randomWinner = PlayMoreOrLessRandomGame(n);

}

else

{

if (n.HasChildren == false)

n.CreateChildren();

UCTNode next = UCTSelect(n); // select a move

if (next == null) //only happens in finished positions and solved nodes - we can start backpropagating ideal result

{

n.IsSolved = true;

if (n.Children.Count == 0) //this is a terminal position - there can be no nodes after it

{

double blackScore, whiteScore;

n.SolvedWinner = n.BoardState.DetermineWinner(out blackScore, out whiteScore);

n.SolvedScore = \_player == 1 ? blackScore : whiteScore;

}

else //this is a non-terminal position for which all possible subsequent moves have been checked

{

if (n.BoardState.ActivePlayer == \_player) //if, for this node, it's this player's turn, then we take the best result

{

bool foundWin = false;

foreach (UCTNode child in n.Children)

{

if (child.IsSolved == false)

throw new ImpossibleException("solved node's child is not solved", "PlaySimulation");

if (child.SolvedWinner == \_player) //if we find a choice that leads to sure win for current player, we immediately take it

{

foundWin = true;

n.SolvedWinner = \_player;

n.Update(1);

return 1;

}

}

//if we don't find a node that leads to current player's victory

n.SolvedWinner = 3 - \_player;

n.Update(0);

return 0;

}

else //if it's enemy's turn on this node, then we take the worst result

{

foreach (UCTNode child in n.Children)

{

if (child.IsSolved == false)

throw new ImpossibleException("solved node's child is not solved", "PlaySimulation");

if (child.SolvedWinner != \_player) //if we find a choice that leads to sure win for enemy, we immediately take it

{

n.SolvedWinner = 3 - \_player;

n.Update(0);

return 0;

}

}

//if we don't find a node that leads to enemy's victory, we assume that this is our winning node

n.SolvedWinner = \_player;

n.Update(1);

return 1;

}

}

}

else

{

randomWinner = PlaySimulation(next);

}

}

int currentPlayerWins = randomWinner == \_player ? 1 : 0;

n.Update(currentPlayerWins); //update node (Node-wins are associated with moves in the Nodes)

return randomWinner;

}

private int PlayMoreOrLessRandomGame(UCTNode n)

{

return \_randomUCT ? PlayRandomGame(n) : PlayLessRandomGame(n);

}

private void ParallelSimulations()

{

while (\_doneSims < \_sims)

{

\_doneSims++;

PlaySimulation(Root);

}

}

// generate a move, using the uct algorithm

public Move GetMove()

{

Move bestMove;

DateTime start = DateTime.Now;

if (\_resetTreeEachTurn == true) //EXPERIMENTAL, HAS NOT BEEN TESTED AND MIGHT NOT WORK

Root = new UCTNode(null, new Move(-5, -5), Root.BoardState.Clone());

Console.WriteLine("Starting Tree size == {0}", Root.MeasureTree());

\_doneSims = 0;

if (Root.Children.Any(\_x => \_x.IsSolved == true && \_x.SolvedWinner == \_player))

{

bestMove = Root.Children.First(\_x => \_x.IsSolved == true && \_x.SolvedWinner == \_player).Position;

}

else

{

for (int i = 0; i < Environment.ProcessorCount; i++)

new Task(ParallelSimulations).Start();

while (\_doneSims < \_sims)

{

Thread.Sleep(100);

}

UCTNode n = GetBestChild(Root);

if (n.Children.TrueForAll(\_x => \_x.Winrate < 0.1))

{

}

if (n == null)

bestMove = new Move(-1, -1);

else bestMove = new Move(n.Position);

}

TimeSpan ts = DateTime.Now - start;

Root.Children.Sort((\_x, \_y) => \_x.Visits.CompareTo(\_y.Visits));

foreach (UCTNode child in Root.Children)

{

Console.WriteLine(child);

}

Root.Children.Shuffle();

Console.WriteLine("Current tree size == {0}, and there are {1} solved nodes", Root.MeasureTree(), Root.CountSolvedNodes());

Console.WriteLine("UCTTurbo-{1} has found move {2}({3},{4}) in {0} after {5} sims", ts, Root.BoardState.ActivePlayer == 1 ? "Black" : "White", Root.BoardState.TurnNumber, bestMove.row, bestMove.column, \_doneSims);

return bestMove;

}

}

}

## GameParameters.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace dotNetGo

{

static class GameParameters

{

public const int BoardSize = 9;

public const double Komi = 6.5;

public const int GameDepth = 500; //maximum turns simulated

public const int TurnTime = 20; //not used

public const Int64 RandomSimulations = 5000; //simulations for random MC

public const Int64 Simulations = 1000; //simulations for proper MC for eyes

public const int UCTSimulations = 25000; //25k+ for proper play

public const int UCTExpansion = 2; //lesser values require more memory but provide deeper search

public const double UCTK = 0.5; //0.44 = sqrt(1/5) - lesser values focus more on successful moves

}

}

## ImpossibleException.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace dotNetGo

{

class ImpossibleException : Exception

{

public string Method { get; private set; }

public ImpossibleException(string message, string methodName) : base(message)

{

Method = methodName;

}

}

}

## IPlayer.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace dotNetGo

{

public interface IPlayer

{

Move GetMove();

bool ReceiveTurn(Move m);

string Name { get; }

}

}

## MonteCarlo.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Runtime.Remoting.Messaging;

using System.Text;

using System.Threading.Tasks;

namespace dotNetGo

{

//using the naive approach - no complex heuristics

internal class MonteCarlo : IPlayer

{

private const int Size = GameParameters.BoardSize;

private Board \_actualBoard = new Board();

[ThreadStatic] private static Move[] \_availableMoves;

[ThreadStatic] private static Board \_testingBoard;

[ThreadStatic] private static Board \_startingTestingBoard;

int GetAvailableMoves(Board b)

{

if (\_availableMoves == null)

\_availableMoves = new Move[Size\*Size+1];

int moveCount = 0;

for (int i = 0; i < Size; i++)

{

for (int j = 0; j < Size; j++)

{

//is on empty space on the board and not a friendly eye

if (b[i, j] == 0 && b.IsEye(i,j) != b.ActivePlayer)

{

\_availableMoves[moveCount++] = new Move(i, j);

}

}

}

return moveCount;

}

public int PlaySimulation()

{

if (\_testingBoard == null)

\_testingBoard = new Board();

\_testingBoard.CopyStateFrom(\_startingTestingBoard);

int turnsSimulated = 0;

while (turnsSimulated < GameParameters.GameDepth && \_testingBoard.IsGameOver() == false)

{

turnsSimulated++;

int moveCount = GetAvailableMoves(\_testingBoard);

Move pass = new Move(-1, -1); //добавить в список возможных ходов пас

\_availableMoves[moveCount++] = pass;

\_availableMoves.Shuffle(moveCount);

for (int i = 0; i < moveCount; i++)

{

if (\_testingBoard.PlaceStone(\_availableMoves[i]) == true)

{

break;

}

}

}

int winner = \_testingBoard.DetermineWinner();

return winner;

}

double GetWinrate(Move move)

{

if (\_startingTestingBoard == null)

\_startingTestingBoard = new Board();

\_startingTestingBoard.CopyStateFrom(\_actualBoard);

if (\_startingTestingBoard.PlaceStone(move) == false)

return -1;

UInt64 sim = 0;

int wins = 0;

while (sim < GameParameters.Simulations)

{

int winner = PlaySimulation();

if (winner != 0)

{

sim++;

if (winner == \_actualBoard.ActivePlayer)

wins++;

}

}

return sim > 0 ? (double)wins / sim : -1;

}

public bool ReceiveTurn(Move m)

{

return \_actualBoard.PlaceStone(m);

}

public string Name

{

get { return "MonteCarlo"; }

}

public Move GetMove()

{

DateTime start = DateTime.Now;

int turnCount = GetAvailableMoves(\_actualBoard);

//most simple logic for the first couple of turns

//reduces required computations and forbids AI from making stupid turns (should not do them anyway)

turnCount = ApplyHeuristics(\_actualBoard, turnCount);

Node[] nodes = new Node[turnCount];

for (int i = 0; i < turnCount; i++)

nodes[i] = new Node(\_availableMoves[i]);

Parallel.For(0, turnCount, (i) =>

{

if (\_availableMoves == null)

\_availableMoves = new Move[Size\*Size + 1];

nodes[i].Winrate = GetWinrate(nodes[i].Pos);

});

double maxWin = -1;

int maxWinIndex = -1;

for (int i = 0; i < nodes.Length; i++)

{

if (nodes[i].Winrate > maxWin && nodes[i].Winrate >= 0)

{

maxWin = nodes[i].Winrate;

maxWinIndex = i;

}

}

DateTime end = DateTime.Now;

TimeSpan ts = end - start;

Move bestMove;

if (maxWin < 0)

{

bestMove = new Move(-1, -1);

}

else

{

bestMove = nodes[maxWinIndex].Pos;

}

Console.WriteLine("Turbo-{1} has found move {2}({3},{4}) in {0} after {5} total sims", ts, \_actualBoard.ActivePlayer == 1 ? "Black" : "White", \_actualBoard.TurnNumber, bestMove.row, bestMove.column, GameParameters.Simulations\*turnCount);

// Console.WriteLine("Coords: {0}", nodes[maxWinIndex].Pos);

return bestMove;

}

private static int ApplyHeuristics(Board board, int turnCount)

{

int j = 0;

int k = 0;

if (board.TurnNumber < 5)

{

while (j < turnCount)

{

Move m = \_availableMoves[j++];

if (m.row > 1 && m.row < 7 && m.column > 1 && m.column < 7)

\_availableMoves[k++] = m;

}

}

else if (board.TurnNumber < 10)

{

while (j < turnCount)

{

Move m = \_availableMoves[j++];

if (m.row > 0 && m.row < 8 && m.column > 0 && m.column < 8)

\_availableMoves[k++] = m;

}

}

else k = turnCount;

return k;

}

}

}

## MonteCarloRandom.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Runtime.Remoting.Messaging;

using System.Text;

using System.Threading.Tasks;

namespace dotNetGo

{

//using the naive approach - no complex heuristics

internal class MonteCarloRandom : IPlayer

{

private const int Size = GameParameters.BoardSize;

private Board \_actualBoard = new Board();

[ThreadStatic]

private static Move[] \_availableMoves;

[ThreadStatic]

private static Board \_testingBoard;

[ThreadStatic]

private static Board \_startingTestingBoard;

int GetAvailableMoves(Board b)

{

if (\_availableMoves == null)

\_availableMoves = new Move[Size\*Size+1];

int moveCount = 0;

for (int i = 0; i < Size; i++)

{

for (int j = 0; j < Size; j++)

{

//is on empty space on the board and not a friendly eye

if (b[i, j] == 0 && b.IsEye(i, j) != b.ActivePlayer)

{

\_availableMoves[moveCount++] = new Move(i, j);

}

}

}

return moveCount;

}

public int PlaySimulation()

{

if (\_testingBoard == null)

\_testingBoard = new Board();

\_testingBoard.CopyStateFrom(\_startingTestingBoard);

int turnsSimulated = 0;

while (turnsSimulated < GameParameters.GameDepth && \_testingBoard.IsGameOver() == false)

{

turnsSimulated++;

Move m = new Move(-1, -1);

do

{

m.row= RandomGen.Next(-1, GameParameters.BoardSize);

m.column = RandomGen.Next(-1, GameParameters.BoardSize);

} while (\_testingBoard.PlaceStone(m) == false);

}

int winner = \_testingBoard.DetermineWinner();

return winner;

}

double GetWinrate(Move move)

{

if (\_startingTestingBoard == null)

\_startingTestingBoard = new Board();

\_startingTestingBoard.CopyStateFrom(\_actualBoard);

if (\_startingTestingBoard.PlaceStone(move) == false)

return -1;

UInt64 sim = 0;

int wins = 0;

while (sim < GameParameters.RandomSimulations)

{

int winner = PlaySimulation();

if (winner != 0)

{

sim++;

if (winner == \_actualBoard.ActivePlayer)

wins++;

}

}

return sim > 0 ? (double)wins / sim : -1;

}

public bool ReceiveTurn(Move m)

{

return \_actualBoard.PlaceStone(m);

}

public string Name

{

get { return "MonteCarlo Random"; }

}

public Move GetMove()

{

DateTime start = DateTime.Now;

int turnCount = GetAvailableMoves(\_actualBoard);

//most simple logic for the first couple of turns

//reduces required computations and forbids AI from making stupid turns (should not do them anyway)

turnCount = ApplyHeuristics(\_actualBoard, turnCount);

Node[] nodes = new Node[turnCount];

for (int i = 0; i < turnCount; i++)

nodes[i] = new Node(\_availableMoves[i]);

Parallel.For(0, turnCount, (i) =>

{

if (\_availableMoves == null)

\_availableMoves = new Move[Size\*Size + 1];

nodes[i].Winrate = GetWinrate(nodes[i].Pos);

});

// for (int i = 0; i < turnCount; i++)

// {

// if (\_availableMoves == null)

// \_availableMoves = new Move[Size \* Size + 1];

// nodes[i].Winrate = GetWinrate(nodes[i].Pos, b);

// }

double maxWin = -1;

int maxWinIndex = -1;

for (int i = 0; i < turnCount; i++)

{

if (nodes[i].Winrate > maxWin && nodes[i].Winrate >= 0)

{

maxWin = nodes[i].Winrate;

maxWinIndex = i;

}

}

DateTime end = DateTime.Now;

TimeSpan ts = end - start;

Move bestMove;

if (maxWin < 0)

{

bestMove = new Move(-1, -1);

}

else

{

bestMove = nodes[maxWinIndex].Pos;

}

Console.WriteLine("StupidTurbo-{1} has found move {2}({3},{4}) in {0} after {5} total sims", ts, \_actualBoard.ActivePlayer == 1 ? "Black" : "White", \_actualBoard.TurnNumber, bestMove.row, bestMove.column, GameParameters.RandomSimulations\*turnCount);

return bestMove;

}

private static int ApplyHeuristics(Board board, int turnCount)

{

int k = 0;

int j = 0;

if (board.TurnNumber < 5)

{

while (j < turnCount)

{

Move m = \_availableMoves[j++];

if (m.row > 1 && m.row < 7 && m.column > 1 && m.column < 7)

\_availableMoves[k++] = m;

}

}

else if (board.TurnNumber < 10)

{

while (j < turnCount)

{

Move m = \_availableMoves[j++];

if (m.row > 0 && m.row < 8 && m.column > 0 && m.column < 8)

\_availableMoves[k++] = m;

}

}

else k = turnCount;

return k;

}

}

}

## Move.cs

using System;

namespace dotNetGo

{

public class Move

{

public int row, column;

public Move(int row, int column)

{

this.row = row;

this.column = column;

}

public Move(Move m)

{

row = m.row;

column = m.column;

}

public override string ToString()

{

return String.Format("{0}:{1}", row, column);

}

public override bool Equals(object obj)

{

Move m2 = obj as Move;

if (m2 == null)

return false;

return row == m2.row && column == m2.column;

}

public override int GetHashCode()

{

return row\*GameParameters.BoardSize + column;

}

}

}

## Node.cs

using System;

namespace dotNetGo

{

class Node {

// public int Wins { get; set; }

// public int Simulations { get; set; }

public Move Pos; // position of move

public double Winrate { get; set; }

public Node(Move m)

{

Pos = new Move(m);

Winrate = 0;

// Wins = 0;

// Simulations = 0;

}

public override string ToString()

{

return String.Format("{0}; {1}", Pos, Winrate);

}

}

}

## Utils.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace dotNetGo

{

static class Utils

{

public static void Shuffle<T>(this IList<T> list)

{

int n = list.Count;

while (n > 1)

{

var k = RandomGen.Next(0, n) % n;

n--;

T value = list[k];

list[k] = list[n];

list[n] = value;

}

}

public static void Shuffle<T>(this IList<T> list, int elementCount)

{

int n = elementCount;

while (n > 1)

{

var k = RandomGen.Next(0, n) % n;

n--;

T value = list[k];

list[k] = list[n];

list[n] = value;

}

}

}

}