Как использовать логические языки чтобы научить компьютер играть в шахматы/шашки

Чтобы понять как научить компьютер играть в шахматы, обратимся к обучению игре в шахматы детей. Детям рассказывают правила поведения фигур, как одна фигуры может ходить, как она может "есть". Также дается информация о том, когда завершается игра(шах и мат). Детям рассказывают какие фигуры более ценные, а какие менее. Почти всю ту же самую информацию мы должны предоставить компьютеру, при этом мы должны дать ему правила обработки ходов и оценки состояния игры. В данной работе я попробую объяснить, как можно научить компьютер играть в шахматы.
Для начала необходимо определить как мы будем хранить фигуры и саму доску. Размер доски 8x8, также каждая клетка может быть разного цвета - черного или белого. Можно хранить всю доску, а для каждой клетки хранить цвет и фигуру, которая там находится или отсутствие фигуры. Можно для каждой фигуры хранить ее позицию и цвет позиции(цвет клетки).
Так как в шахматах все фигуры имеют разный вес, например король имеет больший вес, чем пешка, то для каждой фигуры необходимо хранить еще и ее вес. Нужно обозначить для каждой фигуры ее вес:
Пешка - 1
Конь - 2
Слон - 2
Ладья - 3
Ферзь - 4
Король - 5
Все обозначения пока условны, для понимания "игры". Так как на курсе Логического программирования на моем факультете меня обучали такому логическому языку программирования, как Prolog, то попробуем решить поставленную задачу на этом языке. Prolog основан на языке предикатов первого рода, что значит - для каждого высказывания определяется его истинность или ложность, возможно, с помощью других высказываний. Логика предикатов делает возможным разбить предикат на термы - в нашем случае это объект Пролога - атомы, числа, переменные или структуры.
В нашем случае названия фигур будет атомом, например для пешки - pawn, для ладьи - rook. Их вес - число.
Для хранения координат вместо букв будем использовать цифры, представим, что шахматная доска - это координатная плоскость, то есть для позиции A1 - (1, 1), B1 - (1, 2), C1 - (1, 3), B2 - (2, 2) и так далее. Теперь необходимо определить, какой цвет у той или иной позиции на доске. Попробуем написать простое правило, определяющее цвет позиции на Прологе. Если оба числа четные, или оба числа нечетные, то цвет - черный, если честность различная, то цвет позиции - белый. Цвета будут являться атомами. Попробуем написать такое правило:

```Prolog
color(white, X, Y) :- integer(X), integer(Y), X1 is X mod 2, Y1 is Y mod 2,  (X1 = 0, not(Y1 = 0) ; Y1 = 0, not(X1 = 0)), !.
color(black, _, _).
```

```
В данном правиле также происходит проверка - является ли поданный терм числом, это важно, так как далее мы производим с ним операции. Также используется отсечение, для того, чтобы после определения белого цвета, это правило закончилось и не искались другие варианты цветов. С хранением мы определились, теперь нужно определиться, за какие фигуры(белые или черные) будет играть человек, а за какие - компьютер. Для этого перед началом игры человек должен будет ввести в программу, за какие фигуры он будет играть, в зависимости от его выбора будут определяться развитие игры. Какие же данные нам понадобятся в самом начале игры? Каждая фигура будет представлена списком - ["Имя фигуры", "её вес", "координата 1", "координата 2"]. При старте игры для белых и черных фигур создадим два разных списка(такие индексы у фигур будут, если пользователь играет за белых, если пользователь играет за четных, то списки поменяются местами):<br>
```Prolog
White_list = [ [rook1, 2 ,1, 1], [kNight1, 2, 1, 2], [bishop1, 3, 1, 3], [queen, 4, 1, 4], [king, 5, 1, 5], [bishop2, 3, 1, 6], [kNight2, 2, 1, 7], [rook2, 2 ,1, 8], [pawn1, 1, 2, 1], [pawn2, 1, 2, 2], [pawn3, 1, 2, 3], [pawn4, 1, 2, 4], [pawn5, 1, 2, 5], [pawn6, 1, 2, 6], [pawn7, 1, 2, 7], [pawn7, 1, 2, 8]].

Black_list = [ [rook1, 2 ,8, 1], [kNight1, 2, 8, 2], [bishop1, 3, 8, 3], [queen, 4, 8, 4], [king, 5, 8, 5], [bishop2, 3, 8, 6], [kNight2, 2, 8, 7], [rook2, 2 , 8, 8], [pawn1, 1, 7, 1], [pawn2, 1, 7, 2], [pawn3, 1, 7, 3], [pawn4, 1, 7, 4], [pawn5, 1, 7, 5], [pawn6, 1, 7, 6], [pawn7, 1, 7, 7], [pawn7, 1, 7, 8]].
```
Для написания алгоритма игры, необходимо научить компьютер делать правильные ходы. В нашем случае будем предполагать, что человек вводит верные координаты для движения фигур, то есть для слона человек никогда не введет такие координаты, что слон переместиться на соседнюю позицию. Когда человек будет делать ход - отправлять фигуру(атом) и ее новые координаты компьютеру, мы будем изменять эти координаты в списке и вычислять наилучший ход для компьютера, далее изменить координаты в списке для фигур компьютера и выдавать получившийся ход. Так как перед началом игра человек вводит, за какие фигуры он будет играть(белые или черные), то после запуска, определяется какой список изменять входными значениями(данными от пользователя), а какой список будет изменять компьютер. С начальными данными и с тем, как изменяются позиции мы определились, теперь необходимо понять, какой алгоритм будет использовать компьютер для нахождения наилучшего хода.
```

Для начало нужно понять как двигаются фигуры и описать предикат движения для каждой фигуры.
Правила движений фигур:
Пешки могут ходить только вперед, а "есть" по-диагонали.
Конь ходит по траектории буквы "Г".
Слон ходит по диагонали на любое количество клеток.
Ладья ходит по вертикали и горизонтали на любое количество клеток.
Ферзь может ходить по диагонали, вертикали и горизонтали на любое количество клеток.
Король может ходить на одну клетку в любую по диагонали, вертикали и горизонтали.
В предикате движения на первой позиции будут содержаться: атом фигуры(ее название), текущий список состояний доски, новый список состояний доски. Попробуем реализовать такой предикат для Короля, остальные предикаты движения нужно реализовать аналогично. Для Короля также возможна рокировка с ладьей, но только в том случае, если до этого обе фигуры не ходили. Для этого будем хранить отдельно изменение состояния короля(либо ладьи), и если одна из этих фигур была перемещена, то рокировка не возможна. В предикат перемещения будет подано дополнительно значение с этой информацией. Причем это значение 1 - если происходили изменения, 0 - если нет. Также для безопасной рокировки в предикат подается список фигур противника. Я опишу только длинную рокировку для белых фигур, с короткой нужно поступить аналогично.
Для Короля:

```Prolog
help_move(Z, A) :- append([X], X2, Z), X = [A, _, _, _].

move(king, List, New_list, R, List2) :- append(P, P2, List), help_move(P2, king), append([C], C2, P2), append(Z, [X1, Y1], C), (X2 is X1 + 1, Y2 is Y1, X2 < 9 ; Y2 is Y1 + 1, X2 is X1, Y2 < 9), append(Z, [X2, Y2], Z2), append(Z2, C2, L), append(P, L, New_list), R is 1.

move(king, List, New_list, R, List2) :- append(P, P2, List), help_move(P2, king), append([C], C2, P2), append(Z, [X1, Y1], C), (X2 is X1 - 1, Y2 is Y1, X2 > 0 ; Y2 is Y1 - 1, X2 is X1, Y2 > 0), append(Z, [X2, Y2], Z2), append(Z2, C2, L), append(P, L, New_list), R is 1.

move(king, List, New_list, R, List2) :- append(P, P2, List), help_move(P2, king), append([C], C2, P2), append(Z, [X1, Y1], C), (X2 is X1 + 1, Y2 is Y1 + 1, X2 < 9, Y2
```

< 9 ; X2 is X1 - 1, Y2 is Y1 - 1, X2 > 0, Y2 > 0), append(Z, [X2, Y2], Z2), append(Z2, C2, L), append(P, L, New_list), R is 1.

move(king, List, New_list, R, List2) :- append(P, P2, List), help_move(P2, king), append([C], C2, P2), append(Z, [X1, Y1], C), (X2 is X1 + 1, Y2 is Y1 - 1, X2 < 9, Y2 > 0 ; X2 is X1 - 1, Y2 is Y1 +1, X2 > 0, Y2 < 9), append(Z, [X2, Y2], Z2), append(Z2, C2, L), append(P, L, New_list), R is 1.

pos1(1, 2).
pos2(1, 3).
pos3(1, 4).
pos_help(List, List2) :- member([_, _ , V1, V2], List), (pos1(V1, V2) ; pos2(V1, V2) ; not(pos3(V1, V2)), member([_, _ , V3, V3], List2), (pos1(V3, V3) ; pos2(V3, V4) ; pos3(V3, V4)).
move(king, List, New_list, R, List2) :- R = 0, not(pos_help(List, List2)), append(P, P2, List), help_move(P2, king), append([C], C2, P2), append(Z, [X1, Y1], C), X2 is X1 - 2, append(Z, [X2, Y1], Z2), append(Z2, C2, L1), append([T], T2, P), append(K, [X3, Y3], T), X4 is X3 + 3, append(K, [X4, Y3], T3), append(T3, T2, L2), append(L1, L2, New_list), R = 1.
```

В данных предикатах происходит проверка, не вышли ли мы за границу поля, если вышли, то такой шаг сделать невозможно, следовательно,  предикат будет завершаться неуспехом. В предикате длинной рокировки также проверяется, что между королем и ладьей нет фигур, делается это с помощью вспомогательного предиката pos_help, в котором предикат member проверяет все фигуры и отправляет их координаты в предикаты *pos , в которых хранятся не нужные нам позиции.<br>
Теперь, после того как мы поняли, что собой представляют фигуры, как они могут двигаться, необходимо понять, как будут изменяться наши списки после хода игрока. Когда человек вводит новую позицию для конкретной фигуры, необходимо изменить координаты этой фигуры, для этого нам понадобится предикат append(использовать так, как при перемещении фигур, только вместо процедуры изменения координат, будем просто удалять текущие координаты и добавлять новые). После изменения координат нужно сравнить списки, и если у каких то двух фигур будут одинаковые координаты, то следует удалить фигуру из списка, не "принадлежавшего" компьютеру. Допустим игрок вводит новые координаты - (X, Y) для какой-то фигуры, а список компьютера - List, тогда с если мы найдем такой элемент - member([Z, _, X, Y]) - true, то нам необходимо удалить
```

его, сделать это можно опять же с помощью предиката append. В противном случае нужно оставить список List без изменения.
Мы разобрались со всеми ходами игрока, теперь приступим к алгоритму нахождения наилучшего хода для компьютера. При изучении материала я узнала, что есть такой алгоритм MiniMax - этот алгоритм основан на переборе всех возможных ходов и их оценкой, после перебора выбирается ход с наивысшей оценкой, что означает, что сделав этот ход мы обойдемся минимальными потерями. Такой перебор всех возможных вариантов очень долгий, поэтому его можно оптимизировать, путём отсечений при поиске "совсем" плохих вариантов(алгоритм Alpha-Beta).
Для понимания работы алгоритма нужно разобраться как мы будем давать оценку каждому ходу. Рассмотрим оценку ходов, если мы играем за белых. Нужно оценить положение доски после каждого хода каждой фигуры, что можно сделать путем поиска в дереве состояний, где шагом будет наш предикат движения для фигур. Если после перемещения фигуры в списке противника есть фигура с такими же координатами, то мы должны ее удалить, как было описано выше. Пусть за это удаление отвечает предикат check([List1, List2], [New_list1, New_list2], Q_list), где List1, List2 - это наше текущее положение доски после изменения(после хода), а New_list1, New_list2 - это новое положения доски(оно будет отличаться от старого, если мы съели фигуру противника), Q_list - оценка хода для первого. Значения Q_list меняется, если мы захватили фигуру противника. Если мы захватили короля, то в Q_list перебор должен завершится с максимальным значением. Если мы ничего не захватили, то возвращается значение 0. В итоге, например с помощью findall можно получить список всех возможных ходов со всеми оценками, и выбрать шаг с наибольшей оценкой. Дерево перебора для такого случая будет очень большим, так как среднее количество возможных перемещений одной позиции равно 40, это значит, что если мы хотим оценить все наши шаги на 2 хода вперед, нам нужно перебрать 40x40 позиций, если на 4 шага вперед, то 40x40x40x40. Следовательно, дерево перебора будет расти экспоненциально, поэтому просмотреть дерево до конца невозможно. Как же решить эту проблему?
Первым способом решения такой проблемы является обход с итерационным заглублением, где мы просто ограничиваем глубину дерева, на которую можем зайти.
Второй способ. Алгоритм Alpha-beta. Данный алгоритм основан на отсечении некоторых вариантов, что сокращает перебор. Для этого при переборе нам понадобятся еще две переменные - максимальное значение для черных и максимальное значение для белых. Если в какой-то момент перебора эти два чиста сравняются, то дальше перебирать нет смысла и можно остановиться

на данном этапе перебора. Объяснение в следующем: Представим, что мы играем за белых, и провели оценку одного нашего хода. Тогда после оценки ходов противника(как противник может нам ответить), мы может отсечь некоторые варианты перебора. Допустим мы рассматриваем наш следующий ход, и, если ход противника, который ставит нас в ситуацию хуже, чем самая плохая ситуация после первого нашего хода, то мы отсекаем перебор такого хода.
Еще один вариант развития событий. В шахматах если такое понятие как. Дебюты. Существуют такие дебюты как Староиндийская защита, Ферзевый Гамбит, Волжский гамбит и другие. Для этого требуется создать определенные правила для каждого дебюта. Можно на каждом шаге проверять позиции фигур, и, если противник пошел так, как мы ожидали то мы знаем следующий наиболее выгодный для нас ход, и можем опустить перебор в данном случае, если же противник пошел не так, как мы ожидали, то следует обратиться к перебору и найти выгодный для нас ход.
Аналогично можно сделать с Эндшпилями. Так как эндшпили происходят в конце игры то лишние проверки можно отсечь путем подсчета ходов. Например проверять эндшпили только после 40-45 хода. Для эндшпилей также необходимо создать правила обработки доски.
Проделав всю эту работу я поняла, что с помощью логических языков возможно научить компьютер такой игре, как шахматы. Необходимо задать правила обработки шахматной доски, что является самой сложной задачей, задать параметры для каждой фигуры, правила хода каждой фигуры и правила окончания игры. Можно написать самого легкого противника в виде компьютера, который например, будет продумывать только на один шаг вперед и обучать. Вообще, если задуматься, можно при начале игры предоставить игроку выбор сложности - от этого будет зависеть реализация нашей программы - на сколько ходов мы просчитываем вперед, учитываем ли Дебюты или Эндшпили(и какие учитываем). В прочем, реализаций можно придумать много, я надеюсь основная мысль обработки доски в моем рассказе читателю понятна, дальше все зависит уже от конкретной реализации и идеи.

Литература:
"Программирование шахмат и других логических игр" Е.Корнилов

Данный курсовой проект научил меня обрабатывать списки состояний на языке Prolog. Также я поняла, что прежде чем приступать к реализации программы следует подумать и просмотреть все варианты решений, понять какое лучшее, а потом уже писать код, иначе в процессе(или после) написания можно столкнуться с слишком долгой работой программы. При написании правил на пролог нужно задумываться не только над тем, какой ответ мы в итоге получим, но и какой размер данных нам нужно перебрать для получения ответа. Также я научилась пользоваться отсечениями в языке Prolog, что оказалось очень удобным при написании кода, так как отсечение запрещает откат(бэктрекинг), что сокращает количество переборов. Данный курсовой проект также дал мне понять, что для удобства решения разных типов задач необходимо осваивать новые языки. При написании программы, которая преобразовывала файл GEDCOM в набор фактов на языке Prolog я использовала язык Python, ко на нем получился, как по мне очень коротким, так как там есть удобные средства обработки, если бы я писала на Си, то мой код был бы в несколько раз длиннее. Я сделала вывод, что каждый язык "заточен" под решение определенных задач, поэтому выбор языка для решения поставленной задачи очень важен.

В результате выполнения курсового проекта я научусь обрабатывать данные на языке Prolog, а также с помощью вспомогательного языка преобразовать файл GEDCOM в набор фактов на языке Prolog.