Zuggenerator

Die Aufgabe des Zuggenerators ist es aus einem Spielzustand die Menge aller möglichen (hier pseudo-legalen) Züge zu erzeugen, die der jeweilige Spieler ausführen kann. Intern arbeitet der Zuggenerator sowohl mit Bitboard-Operationen als auch mit Koordinaten-Indizes , die dann über möglichst einheitliche Schnittstellen verarbeitet werden. Die Struktur des Zuggenerators lässt sich abgesehen von den äußeren Schnittstellen am besten durch die Bottom-Up-Perspektive erläutern, die für die Entwicklung auch maßgeblich war. Es existiert eine Hauptroutine *generate_moves_verbose()*, sowie diverse Hilfsroutinen für das Erstellen und Manipulieren der Zuglisten, die einheitliche Schnittstellen für die diversen Zugtypen bereitstellen, die wiederum in eigenen Funktionen implementiert sind.
Der Zuggenerator operiert über alle Phasen des Spiels gleich, und iteriert erst über alle Figurentypen, dann über alle Figuren eines Typen.

Input / Output

Grundsätzlich werden für das Ausführen des Zuggenerators also alle Daten benötigt, die zusammen die Konstellation des Spielfeldes mit den Figuren ergeben, sowie die Information, welcher Spieler am Zug ist. Diese Informationen werden als zwei Argumente der übergeordneten Funktion *generate_moves_verbose(b, player)* übergeben: *b* ist dabei das Dictionary mit den einen Spielfeldzustand codierenden „Quasi-Bitboards“, *player* die als -1/1 codierte Information darüber, ob weiß/schwarz am Zug ist. Die Funktion erzeugt daraus zwei Listen: Die erste enthält alle Spielfeldzustände, die aus pseudo-legalen Zügen hervorgehen, die zweite Liste enthält die dazugehörigen Zugnamen. Eine Besonderheit des Zuggenerators ist hier, dass die erzeugten Züge ihrerseits vollständige Beschreibungen des Folgezustands aus dem eingegebenen Zustand sind, die Struktur also grundsätzlich erhalten bleibt. Der Grund für diese Entscheidung wird unter „Zugaufteilung und Simulation“ erläutert.

Figurenspezifische Zielfeldermittlung

Das Herzstück des Zuggenerators sind Funktionen, die aus einem Spielfeldzustandsdictionary *b* sowie einem Bitboard *bb_from*, ein Bitboard erzeugt, dass alle Felder markiert, auf die die Figur ziehen kann. *bb_from* ist ein Bitboard, das nur die Position der zu ziehenden Figur enthält. Die Funktionen sind dabei nach dem Schema „moves_ZUGTYP_W/B“ benannt. Theoretisch können auch mehrere Figuren des Typs enthalten sein, um beispielsweise eine Felderüberdeckung beider Springer zu erhalten, dabei wird jedoch die Zurordnung uneindeutig, welche Figur auf das jeweilige Feld ziehen kann.

Genau genommen existieren für jeden Zugtypen Funktionen, die unter Annahme einer Position, auf der eine Figur steht alle möglichen Felder generiert, auf die die Figur mit diesem Zugtypen ziehen kann. Da mit Ausnahme der Dame die Zuordnung Zugtyp zu Figurtyp eindeutig ist, sind die Zugermittlungsfunktionen häufig entsprechend nach den Figurtypen benannt. Da die Dame die gleichen Züge ausführen kann wie ein Läufer oder ein Turm, können deren Zuggeneratoren aufgerufen und die Ergebnisse mit einem logischen „OR“ zusammengefasst werden.
Für jeden Zugtypen existiert jeweils eine Funktion pro Farbe (Schwarz/Weiß), da einerseits zwischen eigenen/gegnerischen Figuren unterschieden werden muss, andererseits die Bauern nur in eine Richtung ziehen können. Die Existenz zweier verschiedener Funktionen erspart dabei eine Fallunterscheidung, was bei einem Aufruf des Zuggenerators somit bis zu 16 Branchingoperationen entspricht. Die Entscheidung welche der beiden Funktionen aufgerufen wird, wird somit eine Ebene darüber getroffen, was bei 6 Figurentypen eine Nettoersparnis von 10 Fallunterscheidungen pro Aufruf bedeutet.
Für die Figurentypen Bauer, König und Springer werden auf den entsprechenden Bitboards logische Operationen und Shifts genutzt, für die Figuren Turm, Läufer und Dame werden iterativ die möglichen Richtungen und Felder durchgegangen, bis auf ein Hindernis getroffen wird. Die Funktionen bekommen auch hier das Bitboard mit der aktuellen Position der Dame übergeben, behandeln diese aber als Turm oder Läufer.
Da Bauern unterschiedliche Zugtypen beispielsweise beim Schlagen als bei leisen Zügen durchführen können, existieren auch jeweils separate Funktionen dafür (*moves_attack_pawn()*, *moves_quiet_pawn()*).

Zugaufteilung und Simulation

Um die Zielfelder aller Figur zu ermitteln, existieren für jeden Figur-/Zugtypen Funktionen „*gen_moves_ZUGTYP()*“, die für jede Figur der Art aufgerufen werden, und zu einer Liste zusammengefügt werden, wobei jede Zeile sowohl das Bitboard *bb_from* sowie das Bitboard aller möglichen Zielfelder enthält. Diese werden dann einheitlich mit der Funktion *gen_capture_quiet_lists_from_all_moves()* verarbeitet:
Zeile für Zeile werden mit *split_capture_quiet()* die Bitboards mit den Zielfeldern zuerst in zwei Bitboards vollständig zerlegt, die die Züge in schlagende oder stille Züge einteilen. Das darauf folgende Verfahren gilt dann analog für schlagende/stille Züge. Die Bitboards werden weiter vollständig zu einer Liste von Bitboards *bb_to* zerlegt, sodass auf jedem Bitboard nur noch ein Zielfeld markiert ist. An diesem Punkt werden aus der Summe aller Züge für eine Figur also alle einzeln ausgewählt, die dann in dem Zugsimulator simuliert werden. Das Ergebnis ist eine Liste gleicher Struktur, die nun nicht mehr die Informationen Ausgangs-/Zielfeld enthält, sondern den vollständigen Spielfeldzustand der aus dem Zug folgt.
Hintergrund für diese Designentscheidung war die Annahme, dass für die Bewertung eines Zugs die Historie des Spielfelds keine Rolle spielt, sondern ausschließlich der aktuelle Zustand. Weiter wurde angenommen, dass die Namen der Spielzüge später nur bei den Schnittstellen ein Mal pro Zug außerhalb der eigentlichen Engine benötigt werden würden, weswegen diese nachträglich

separat durch eine Differenzbetrachtung der Spielfelder errechnet werden konnten, was gegenüber einer sofortigen Berechnung Zeit sparen sollte. Das einfache Generieren von Paaren aus *bb_from* und *bb_to* wäre zwar möglich gewesen, jedoch hätte in dem Fall entweder der alte Spielfeldzustand mit abgespeichert werden müssen, oder beim Traversieren des Suchbaums jede Zugserie jedes Mal aufs Neue simuliert werden müssen. Beim Speichern des alten Spielfeldzustands zusammen mit *bb_from* und *bb_to* in einem Knoten würde der aktuelle Zustand also eine Ebene tiefer auch auftauchen. Insgesamt wären die Informationen somit redundant im Suchbaum, weswegen sich für das alleinige (spätere) Abspeichern von den neuen Spielfeldzuständen entschieden wurde.
Nach dem iterieren über alle Figuren des Figurentyps - hier zum Beispiel über alle Bauern - wird die erzeugte Liste von Listen aller möglichen Züge jedes Bauern zu einer Liste aller Bauernzüge abgeflacht. Am Ende existieren somit nur noch eine Liste aller schlagenden Bauernzüge (*move_list_capture_flat*), und eine Liste aller leisen Bauernzüge (*move_list_quiet_flat*).
Als sich später für die *Transpositiontables* für jeden Zug *unique identifiers* benötigt wurden, wurde die Idee der nachträglichen Generierung von Zugnamen im *Post processing* verworfen und analog zum Zugsimulator ein Zugnamensimulator implementiert, sodass neben den Listen für schlagende/leise Zugfolgezustände schlagende/leise Listen mit den dazugehörigen Namen (*name_list_capture_flat* bzw*. name_list_quiet_flat*) erzeugt werden.

Sammeln der Züge aller Figurtypen und Vorsortierung

Da nun die Möglichkeit besteht alle möglichen Folgezustände für Züge bestimmter Art zu erzeugen, können auf der höchsten Abstraktionsebene (die Hauptroutine des Zuggenerators *generate_moves_verbose()*) diese Listen nacheinander für alle dieser Zugtypen erstellt werden. Anschließend werden die Listen so zusammengefügt, dass der Zuggenerator nur noch zwei Listen ausgibt: *move_list*, die alle Züge enthält sowie *name_list*, die die dazugehörigen Zugnamen enthält. Die Reihenfolge in der die Züge hier in die Liste eingefügt werden entspricht einer Vorsortierung, da die Züge der Reihe nach in den Suchbaum eingepflegt werden.
Die Sortierung kann für die Baumsuche später von großer Relevanz sein, da bei optimaler Sortierung möglichst viele Alpha/Beta-Cutoffs erzielt werden können. Die Frage nach einer guten Vorsortierung schien jedoch schwierig zu beantworten zu sein: Intuitiv sind schlagende Züge erstrebenswert, da sie die eigenen Gewinnchancen verbessern, jedoch kommt es im Schach permanent zu gedeckten Spielfiguren. Das Schlagen einer gedeckten Spielfigur wird bei Priorisierung schlagender Züge somit immer wieder untersucht, obwohl sich die Deckungssituation der Figur teilweise über viele Züge nicht verändert.
Generell ist also die Sortierung die beste, die die Figur mit der höchsten Wahrscheinlichkeit bewegt zu werden (bester Zug) an die erste Stelle sortiert. Bei Performanceproblemen der Baumsuche hätte man hier die Reihenfolge entsprechend optimieren können, schlussendlich wurden jedoch erst die schlagenden Züge und dann die leisen Züge in die Liste sortiert.

Zugsimulator

Dem Zugsimulator ist in der Funktion *make_move(b_old, bb_from, bb_to)* implementiert. Grundsätzlich muss für das Erzeugen des neuen Spielfeldzustands *b_new* aus dem alten (*b_old*) zwischen schlagenden und leisen Zügen unterschieden werden: Bei leisen Zügen werden alle Einträge auf allen Bitboards die mit *bb_from* Überschneidungen haben an der Stelle mittels *XOR* gelöscht, während die neue Position *bb_to* in diesem Fall durch logisches OR eingetragen wird. Bei schlagenden Zügen müssen alle Bitboards die Überschneidungen mit der neuen Position *bb_to* haben zusätzlich so aktualisiert werden, dass der Eintrag analog zu leisen Zügen an dieser Stelle entfernt wird. Die Implementierung auf diese Art und Weise stellt sicher, dass zusätzliche Bitboards die später in den Spielfeldzustand aufgenommen und weitere Informationen speichern könnten automatisch richtig aktualisiert werden können, ohne das eine weitere Anpassung des Codes notwendig ist.
Eine Besonderheit ist die Damenumwandlung, die einen eigenen Zugsimulator mit der Funktion *make_move_pawn_to_queen(b_old, bb_from, bb_to)* zum Umgehen von unnötigen Fallunterscheidungen hat. Dabei werden nach dem Ausführen des normalen Zugsimulators die Felder der Bauern und der Damen mit dem Bitboard der neuen Position *bb_to* mittels XOR aktualisiert; sodass der Bauer an der Stelle verschwindet und eine Dame entsteht.

Zugnamensimulator

Für den Zugnamesimulator *make_move_name(b_old, bb_from, bb_to, short = False)* werden wie beim Zugsimulator Überschneidungen von Bitboards ausgenutzt: Ist beispielsweise die Überschneidung des Bitboards *bb_from* mit dem statischen Bitboard für die Reihe 3 nicht leer, so lässt sich daraus schließen, dass die Figur auf der Reihe 3 steht. Iteriert man so über alle Linien und Reihen kann man sowohl für *bb_from* als auch *bb_to* einen String für die Position vor sowie nach dem Zug generieren. Nach der gleichen Logik kann die gezogene Figur, sowie der ziehende Spieler ermittelt werden. Ebenso wurde zwischenzeitlich die vollständige Notation mit zusätzlicher Kennzeichnung eines schlagenden Zugs mittels eines „x“ implementiert, was für einheitliche interne Verarbeitung bei den *Transposition tables* jedoch wieder verworfen wurde. Analog wurde die Kurzschreibweise defaultmäßig deaktiviert, bei der Bauern als Figurtyp nicht explizit erwähnt werden.
Ein fertiger Zugname wird dann nach dem Schema „Figurtyp + Linie vorher + Reihe vorher + Linie nachher + Reihe nachher“ zusammengesetzt, wobei der Figurtyp gemäß der etablierten Notation für weiße Spielzüge groß und für schwarze Spielzüge klein geschrieben wird.

Baumsuche

Die Baumsuche basiert auf iterativer Tiefensuche in Verbindung mit *Aspiration Windows* und *Principal Variation Search.* Die Umsetzung der *Principal Variation Search* entspricht dem Pseudocode aus dem Wikipedia-Artikel (https://de.wikipedia.org/wiki/Alpha-Beta-

Suche#Principal-Variation-Suche). Die Aspiration Windows kommen ab der zweiten Iteration der Tiefensuche zum Einsatz, da in der ersten Iteration noch kein Erwartungswert verfügbar ist. Als initiales Fenster um den Erwartungswert wird dabei in beide Richtungen der Wert eines halben Bauern gewählt, wobei sich das Fenster bei Fehlschlag exponentiell in die Richtung erweitert, in der der Fehlschlag auftrat. Die Konstante die auf den bisherigen Rand angeschlagen wird ist dabei anderthalb Bauern und wird um die Anzahl der Fehlschläge in der entsprechenden Richtung potenziert. Die Idee von exponentiell wachsenden Fenstern wurde ebenfalls aus dem Wikipediaartikel übernommen.
Eine Besonderheit bei der Suche stellt die Verwendung einer rudimentären Zeitabschätzung für die Dauer der nächsten Tiefeniteration dar. Dafür wird die Zeit jeder Integration gemessen und mit der Funktion *time_expected_next()* die Zeitabschätzung berechnet. Bleibt am Ende einer Iteration noch mehr Zeit für die Suche als durch die Zeitabschätzung veranschlagt, wird die Berechnung einer weiteren Ebene begonnen. Dadurch kann Zeit gespart werden, da insbesondere in der letzten Iteration viel Zeit aufgewendet wird, die bei nicht vollständig untersuchten Ebenen minderwertige Ergebnisse liefern kann, sollte die Suche vorzeitig abgebrochen werden. Eine weitere Besonderheit ist das Speichern der untersuchten Knotenwerte in den Knoten, was theoretisch eine Betrachtung der Knotenwerte über die Suchtiefe ermöglicht. Am Ende der Suche wird dadurch nicht nur ein einzelner Zug nach oben propagiert, stattdessen kann aus allen Züge der ersten Ebene einer mit der maximalen Bewertung ausgewählt werden, sofern mehrere existieren. Ist das der Fall wird davon einer zufällig ausgewählt, was dazu führt, dass beispielsweise nicht immer die gleiche Spieleröffnung gewählt wird.
Da bei der Alpha-Beta-Suche Cutoffs auftreten musste bei dieser Implementierung beachtet werden, dass durch Cutoffs abgeschnittene Knoten nicht aktualisiert werden, und somit zum Beispiel zu positiv bewertete Knoten aus der letzten Iteration die Suche stören könnten. Deshalb wird bei jeder Iteration eine *Deepcopy* des Suchbaums geladen, sodass alle Knoten wieder unbewertet sind.