36. Bundeswettbewerb Informatik
Aufgabe 5
Bauernopfer
Dokumentation

Kamal Abdellatif

27. November 2017

Treibjagd

Ziel der Bauern ist es, die Zugmöglichkeiten des Turms einzugrenzen. Steht dem Turm viel freie Fläche zur Verfügung, so kann er nur schwer gefangen werden, da viele Ausweichmöglichkeiten vorhanden sind. Wird der Turm jedoch nach und nach in seinen Möglichkeiten eingegrenzt, so kann er schließlich immer gefangen werden. Die folgend beschriebene Strategie versucht, die Möglichkeiten des Turms ausgänglich von einer gegebenen Anfangsstellung zu minimieren. Damit das Programm mögliche Vorstöße der Bauern bewerten kann, wird eine quantitative Definition der Zugmöglichkeiten des Turms benötigt.

> Die *Zugmöglichkeiten* des Turms einer gegebenen Stellung sind die Menge aller Felder, die der Turm in einer endlichen Anzahl von erlaubten Zügen ausgehend von seiner aktuellen Position erreichen kann.

Erlaubte Züge des Turms sind alle Züge, die horizontal oder vertikal von einer Startposition zu einer Endposition verlaufen, ohne dabei einen Bauern zu überspringen. Die Endposition darf dabei kein Feld sein, welches von einem Bauern besetzt oder bedroht ist. Ein *bedrohtes Feld* ist ein Feld, das in einem Zug von einem Bauern erreicht werden kann. Der Turm gilt als gefangen, wenn er keine Zugmöglichkeiten mehr besitzt.

Würde der Turm auf ein solches Feld ziehen, so würde er sofort geschlagen werden können. Es kann davon ausgegangen werden, dass der Turm optimal spielt. Daher wird der Turm nie ein bedrohtes Feld besuchen.
Die möglichen erreichbaren Felder sind rekursiv definiert:

Rekursionsanfang
Die aktuelle Position des Turms ist erreichbar.

Rekursionsschritt
Ist ein Feld erreichbar, so sind es alle Felder, auf die der Turm von diesem Feld aus ziehen könnte, auch.

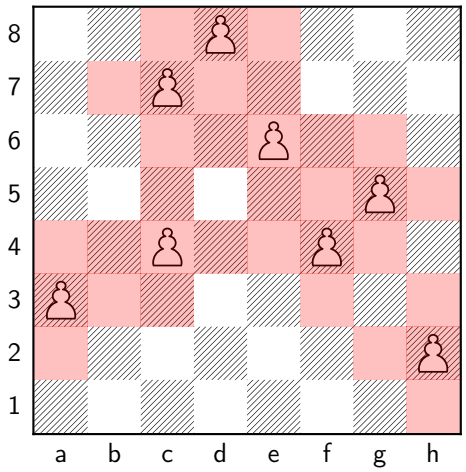

Abbildung 1: Bedrohte Felder (rot)

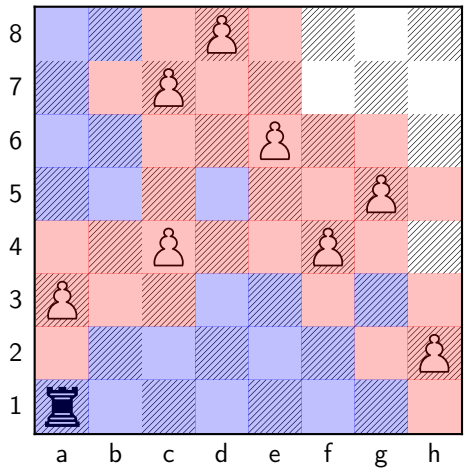

Abbildung 2: Erreichbare Felder (blau)

Die Bauern folgen einer Einschnürungs-Strategie. Dabei sollen die Zugmöglichkeiten stetig verringert werden. Demzufolge ist ein Bauernzug, der die Anzahl von erreichbaren Feldern vergrößert, nicht Teil der Strategie. Ein solcher Zug wird als *Öffnung* bezeichnet. So kann ein Zug auf den ersten Blick harmlos erscheinen, dem Turm aber ein Schlupfloch verschaffen. Ein Beispiel dafür wäre hier der Zug e6→e5. Solange keine Öffnung auftritt, gilt die Bauernstellung als *geschlossen*.

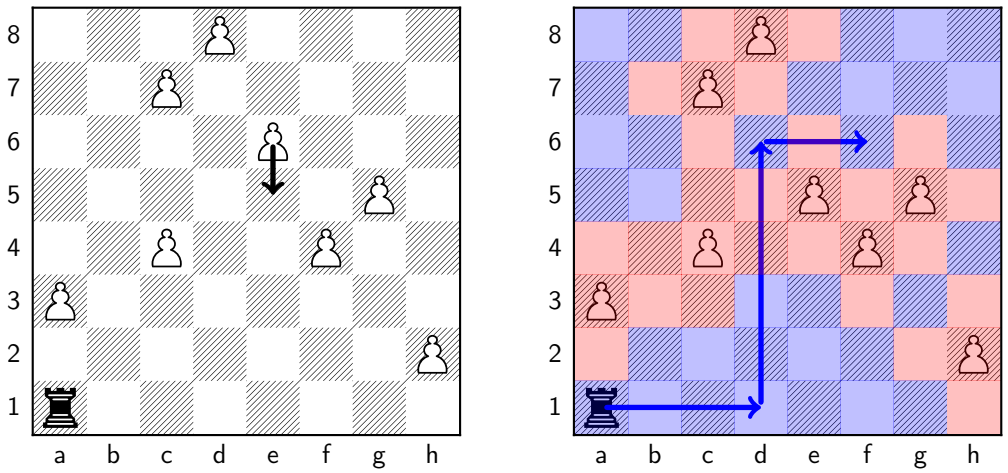

Abbildung 3: Öffnung der Bauernstellung

Die Anfangsstellung wurde so gewählt, dass von Anfang an die Möglichkeiten des Turms stark reduziert sind. Wenn die Bauern eine geschlossene Stellung halten, so müssen sie am Anfang das Brett zweiteilen. Da der Turm nach Aufstellung der Bauern sich in das Feld setzen kann, hat der die Möglichkeit, den größeren und somit günstigeren Teil für sich zu wählen. Um diese Wahl so ungünstig wie möglich zu machen, werden beide Hälften gleichgroß gewählt, da der Turm so keinen Vorteil durch die Wahl der Hälfte erhält. Unter diesen Gesichtspunkten wurde folgende Startposition gewählt

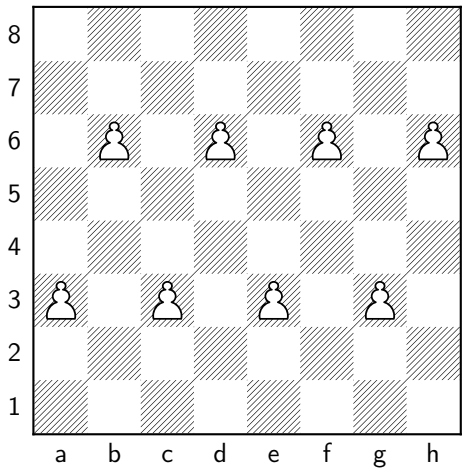

Abbildung 4: Startposition

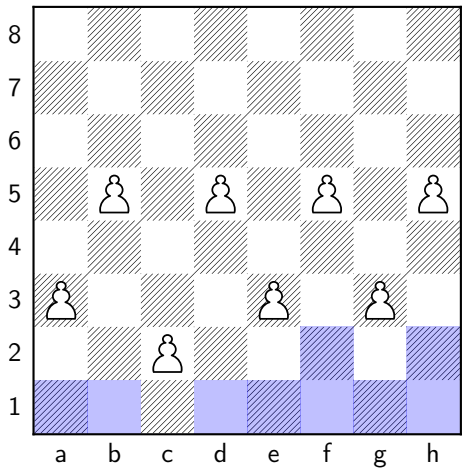

Abbildung 5: Bauernstellung nach 5 Zügen

Das Programm beginnt darauf hin, nach jedem Zug des Turms jeden möglichen Bauernzug zu betrachten. Es wählt dabei den Bauernzug, welcher die erreichbaren Felder des Turms am meisten verringert. Obwohl dieses „greedy“-Vorgehen kurzsichtig erscheinen mag, ist es dennoch sehr effizient. Mit dem ersten Zug wählt der Turm eine Seite. Auf Grund der Symmetrie wird im Folgenden ohne Beschränkung der Allgemeinheit die untere gewählt. Die ersten 5 Züge der Bauern waren unabhängig vom Verhalten des Turms. Die Bauern auf der 6. Reihe rücken zuerst vor, da diese jeweils mit einem Zug nach vorne ein erreichbares Feld des Turms eleminieren. Dies trifft nicht auf die vorderen Bauern zu, da diese neue Felder hinter sich öffnen würden. Diese Öffnungen sind im 5. Zug abgedeckt, da nun die 5. Reihe besetzt ist. Die Mittelbauern können durch vorrücken 4 Felder eleminieren, die Randbauern nur 3. Demnach zieht immer zuerst einer der Mittelbauern. Je nach Verhalten des Turms findet dieser sich innerhalb der nächsten 5 Züge mit einer Außnahme in einer der folgenden vier Situationen wieder. Der Turm besitzt keine Zugmöglichkeiten mehr, außer in der Zwickmühle zu verharren.

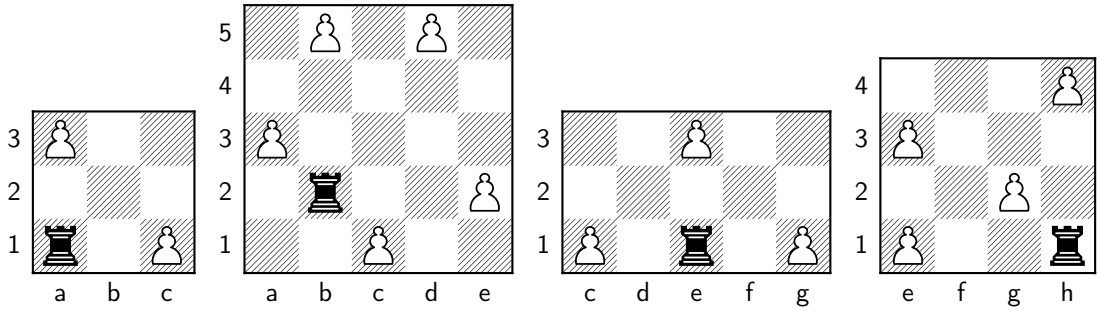

Abbildung 6: Der Turm kommt ins Schwitzen

Für jede dieser Stellungen ist der Turm in maximal 3 Zügen und somit in insgesamt 12 Zügen geschlagen. Die Außnahme ist das optimale Verhalten des Turms, welches im 12. Zug folgend endet[1]. Im 13. Zug ist der Turm garantiert geschlagen.

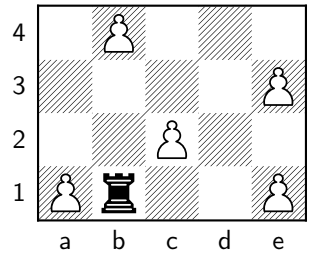

Abbildung 7: Optimales Ende

[1]Der gesamte optimale Spielverlauf ist im Anhang zu finden

Wenn das Programm die oben beschriebene Strategie fehlerfrei befolgt, so wird der Turm immer in maximal 13 Zügen geschlagen.

Treibjagd mit 7 Bauern

Es ist 7 Bauern nicht möglich, den Turm zu fangen, solange sich dieser optimal verhält. Mit sieben Bauern muss es nach Schubfachprinzip mindestens eine Reihe und mindestens eine Linie geben, welche von keinem Bauern besetzt ist. Im ersten Zug setzt sich der Turm auf den Schnittpunkt von freier Reihe und Linie.

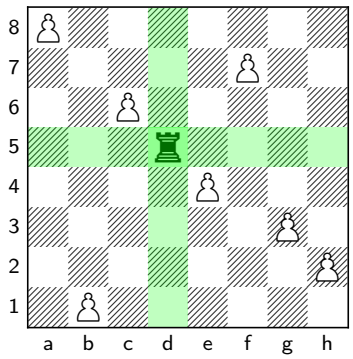

Abbildung 8: Freie Reihe (5) und Linie (d)

Im weiteren Verlauf des Spiels wird der Turm diesem Schnittpunkt folgen. Da Bauern sich nur horizontal oder vertikal bewegen können, kann sich mit jedem Bauernzug *entweder die freie Linie oder freie Reihe* ändern. So liegen neuer und alter Schnittpunkt immer entweder auf der gleichen freien Reihe oder Linie. Das bedeutet, dass der Turm den neuen Schnittpunkt immer innerhalb eines Zuges erreichen kann.

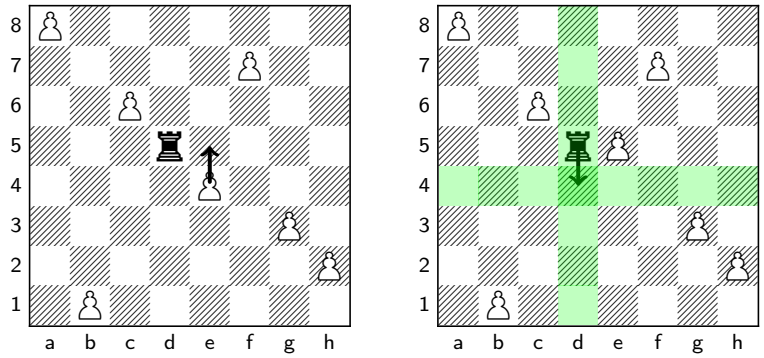

Abbildung 9: Reaktion des Turms auf Verschiebung des Schnittpunks

Per Induktion wurde also gezeigt, dass es dem Turm immer gelingt, auf dem Schnittpunkt einer freien Linie und Reihe zu stehen. Der Turm besitzt dadurch immer mindestens 15 Zugmöglichkeiten, und kann demnach nie gefangen werden.

Allgemeiner Fall

Sei $\mathcal{L}$ die Menge aller Tupel $(k,\ \ell) \in \mathbb{N}^* \times \mathbb{N}^*$, für die der Turm gefangen werden kann. Da maximal alle Bauern ziehen können, gilt $\ell \leq k$. Kann der Turm für eine Einstellung $(k,\ \ell)$ gefangen werden, so ist dies auch für alle Einstellungen mit größerem k und oder größerem ℓ möglich. Ist es für diese Einstellung jedoch nicht möglich, so ist es auch für kleinere k und oder ℓ nicht möglich.

$$(k,\ \ell) \in \mathcal{L} \ \Rightarrow\ \forall k', \ell'\ \ \big((k' \geq k) \wedge (\ell' \geq \ell) \Rightarrow (k',\ \ell') \in \mathcal{L}\big)$$
$$(k,\ \ell) \notin \mathcal{L} \ \Rightarrow\ \forall k', \ell'\ \ \big((k' \leq k) \wedge (\ell' \leq \ell) \Rightarrow (k',\ \ell') \notin \mathcal{L}\big)$$

Wie bereits in den vorherigen Aufgabenteilen gezeigt, ist es möglich, den Turm für $k = 8,\ \ell = 1$ zu fangen, jedoch nicht für $k = 7,\ \ell = 1$. Demnach wissen wir, dass es für alle Tupel mit $k \geq 8$ möglich ist, aber nie für $k \leq 7,\ \ell = 1$.

$$\forall k, \ell\ ((k \geq 8) \Rightarrow (k,\ \ell) \in \mathcal{L}) \qquad (\ell \in \mathbb{N}^* \Rightarrow \ell \geq 1)$$
$$\forall k\ ((k \leq 7) \Rightarrow (k,\ 1) \notin \mathcal{L})$$

Es bleibt der Bereich $k \leq 7,\ \ell \geq 2$ unbetrachtet. Dazu werden nun die zwei Fälle $(3,\ 2)$ und $(4,\ 2)$ genauer untersucht.
Der Turm kann nie gefangen werden, wenn es ihm immer gelingt, die Bauernstellung zu durchstoßen, da er so, sobald er sich auf der kleineren Hälfte befindet, auf die größere wechseln kann. Dies wäre ein sich endlos wiederholender Zyklus, in dem der Turm nie gefangen werden kann.

Ist es dem Turm immer möglich, innerhalb einer endlichen Anzahl von Zügen seine erreichbaren Felder auf die andere Hälfte zu erweitern, kann er nie gefangen werden.

Zuerst wird die Einstellung $k = 3,\ \ell = 2$ betrachtet. Eine mögliche Anfangsstellung für 3 Bauern ist in Abbildung 10 zu sehen. Versucht der Turm, auf die andere Seite zu gelangen, so muss er die Bauernreihe über eine freie Linie durchstoßen. Versucht der Turm, eine freie Linie zu erreichen, kann er zunächst von den Bauern geblockt werden.

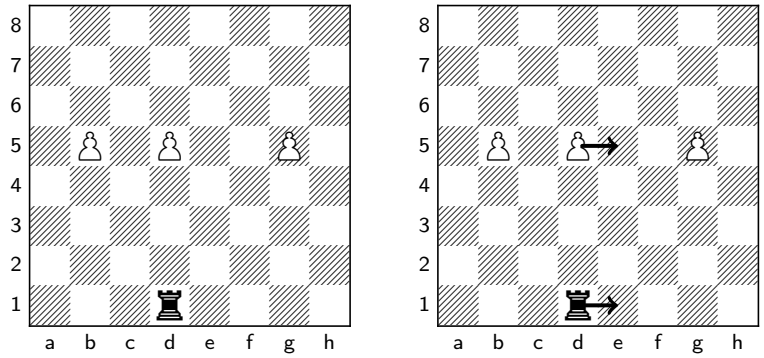

Abbildung 10: Anfangsstellung und Block

Das Brett wird in linke (a-d) und die rechte (e-h) Hälfte aufgeteilt. Nach Schubfachprinzip muss es immer eine Hälfte geben, die nur einen oder keinen Bauer besitzt. Betrachtet wird ohne Beschränkung der Allgemeinheit die rechte Hälfte. Der Turm kann nur auf Linien durchstoßen, die nicht innerhalb eines Zuges von einem Bauern geblockt werden können. Ist die Hälfte leer, so trifft dies auf die Linien f, g und h zu[2]. Für alle 4 möglichen Linien, die der einzelne Bauer in der rechten Hälfte einnehmen kann, ist es dem Turm möglich, eine Öffnung zu erzwingen. In Abbildung 11 sind alle vier Zugabläufe als Spalten parallel dargestellt. Reihen werden der Anschaulichkeit halber vernachlässigt. Der Turm setzt im Bild zuerst.

[2] e könnte noch von einem Bauern der linken Hälfte geblockt werden.

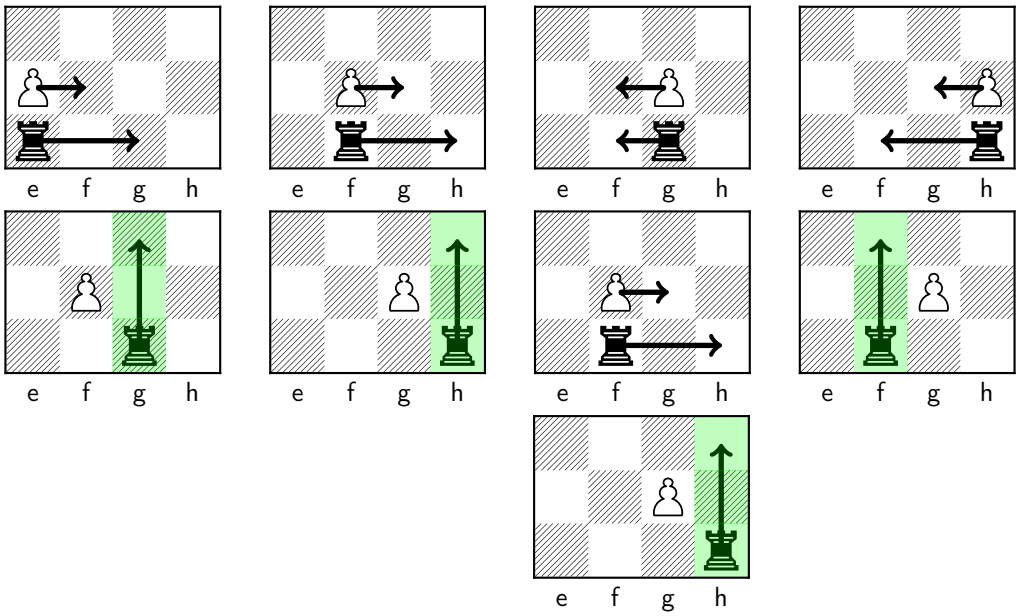

Abbildung 11: Durchstoß des Turms

Der Turm kann demnach immer die Bauernstellung durchstoßen, sodass er nie gefangen werden kann. Folglich befindet sich $(3,\ 2)$ nicht in $\mathcal{L}$. Diese Argumentation wurde unabhängig von den verbleibenden Zügen der Bauern geführt, da der einzeln stehende Bauer trotzdem nur einmal pro Turmzug ziehen kann. Es trifft demnach auch auf $(3,\ 3)$ zu.

Für die Einstellung $k = 4,\ \ell = 2$ kann diese Schwachstelle nicht mehr ausgenutzt werden, da nun beide Hälften mit jeweils zwei Bauern besetzt sind. Jeder der 4 Bauern wird einem Linienpaar (ab, cd, ef, gh) zugeordnet. Jeder Bauer bewegt sich nur noch auf den zwei ihm zugeteilten Linien. So ist eine Linie des Paars immer geblockt, während die andere im nächsten Zug geblockt werden kann. So kann, egal auf welche Linie der Turm zieht, diese immer in maximal einem Zug geblockt werden. Da zum Blocken nur maximal ein Bauernzug benötigt wird, können der bzw. die Verbleibenden der zwei Züge genutzt werden, um mit der Formation in Richtung des Turms vorzurücken.

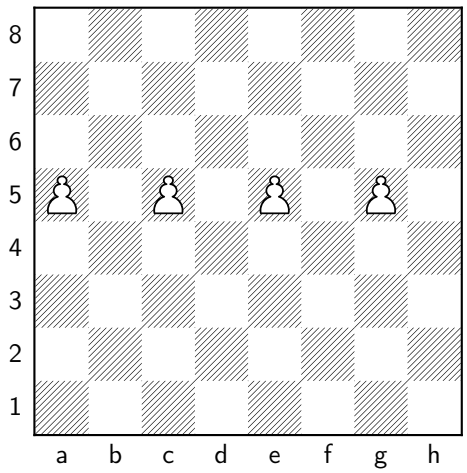

Abbildung 12: Startposition mit 4 Bauern

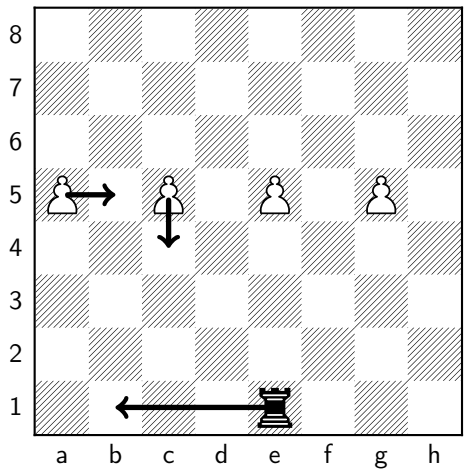

Abbildung 13: Blocken und Vorrücken

Verfolgen die Bauern diese Strategie, und können regelmäßig vorrücken, so findet sich der Turm in kurzer Zeit auf der Grundlinie wieder, während die Bauern die darüber liegende Reihe besetzen. Setzen nun Bauern auf die Grundlinie, so müssen diese nicht weiter vorrücken, sondern können den Turm nun seitlich in seiner Bewegungsfreiheit eingrenzen.

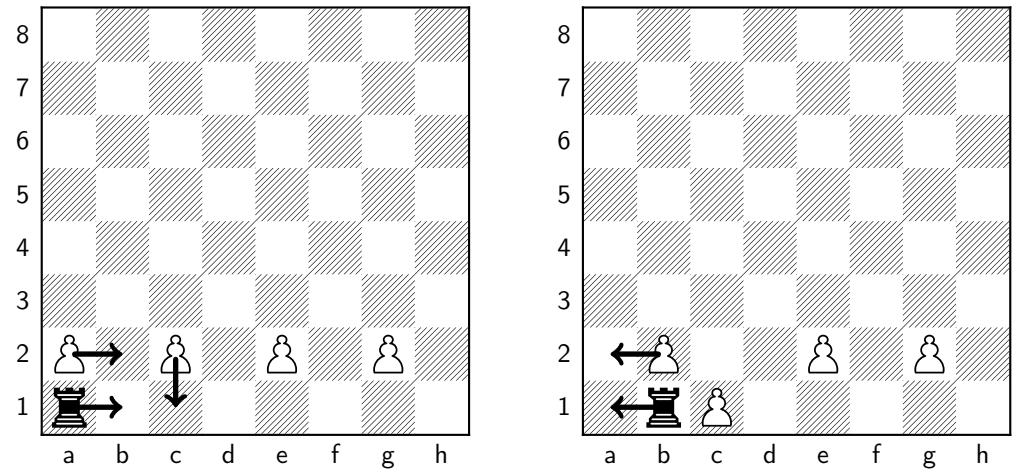

Abbildung 14: Endstadium: Seitliche Eingrenzung

Es kann jedoch passieren, dass der Turm sich auf einen Bauern konzentriert, sodass dieser immer Blocken muss. Damit, dass dieser Bauer so nicht vorziehen kann, könnte der Turm versuchen sich ein Schlupfloch zu machen. Wie eine solche Stellung vor dem Endstadium aussehen könnte, ist in Abbildung 15 zu sehen.

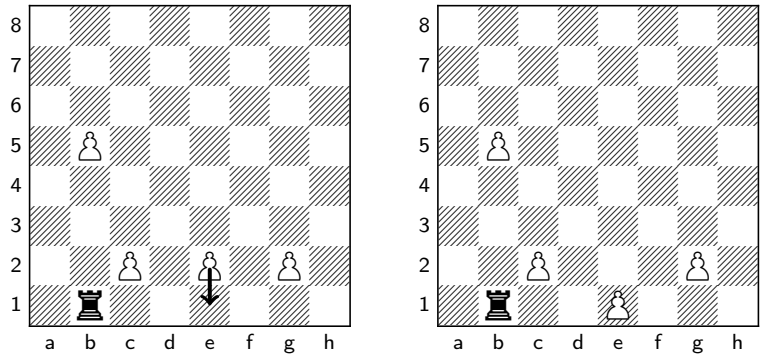

Abbildung 15: Öffnung der Stellung durch Zugzwang

Der Bauer auf e2 beginnt, seitlich einzugrenzen. Der drohende Zug Tb1-b3 kann jedoch von den Bauern gekontert werden.

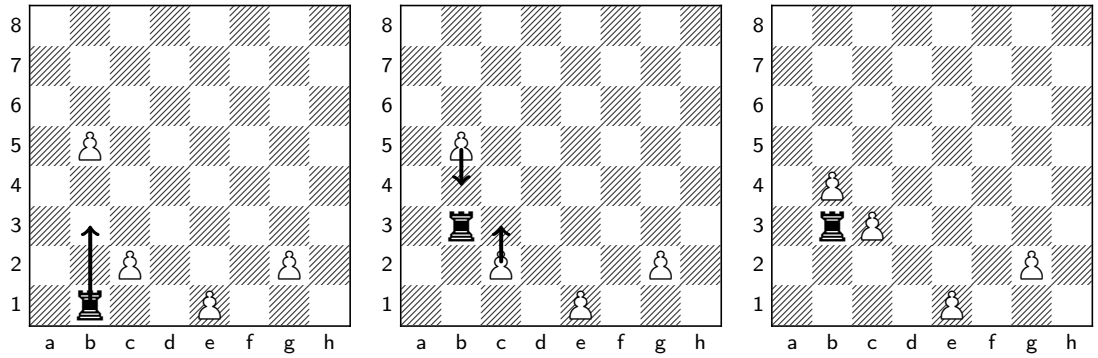

Abbildung 16: Konter des Fluchtversuchs

Die maximale Tiefe dieses Einschnittes in die Formation können zwei Felder sein, wie im obigen Beispiel. Da das erste Feld von dem noch nicht vorgerückten Bauer, hier b5, bedroht wird, bleibt dem Turm nur ein Feld als Schlupfloch. Unter Verwendung der Züge, die vorher für das Vorrücken übrig geblieben waren, können so die Bauern der benachbarten Linienpaare anrücken, um so diese Öffnung zu bedrohen, sodass der Turm zurückziehen muss. Der Turm kann die Formation nie durchbrechen und wird immer gefangen.
Demnach befindet sich $(4,\ 2)$ in $\mathcal{L}$.

Es wurde gezeigt dass $(3,\ 2),\ (3,\ 3) \notin \mathcal{L}$ und $(4,\ 2) \in \mathcal{L}$. So können die Lücken geschlossen und $\mathcal{L}$ in geschlossener Form dargestellt werden. Für $k \leq 3$ wird der Turm immer gefangen. Für $4 \leq k \leq 7$ kann der Turm nur für $\ell = 1$ gefangen werden. Für alle $k \geq 8$ ist der Turm immer zu fangen.

$$(3,\ 3) \notin \mathcal{L} \ \Rightarrow\ \forall k, \ell\ ((k \leq 3) \Rightarrow (k,\ \ell) \notin \mathcal{L}) \qquad (\ell \leq k \leq 3)$$
$$(4,\ 2) \in \mathcal{L} \ \Rightarrow\ \forall k,\ \ell\ ((k \geq 4) \wedge (\ell \neq 1) \Rightarrow (k,\ \ell) \in \mathcal{L})$$

Es kann mit Ausnahme folgender Tupel für alle $(k,\ \ell)$ der Turm gefangen werden.

$$\overline{\mathcal{L}} = \{(1,\ 1),\ (2,\ 1),\ (2,\ 2),\ (3,\ 1),\ (3,\ 2),\ (3,\ 3),\ (4,\ 1),\ (5,\ 1),\ (6,\ 1),\ (7,\ 1)\}$$

Beliebig lange Schritte

> „Was wäre, wenn die Bauern nicht nur Schritte von je einem Feld, sondern n Feldern machen könnten? “
>
> – *Kamal*

Es sei $\mathcal{M}$ die Menge aller Tupel $(k,\ \ell,\ n) \in (\mathbb{N}^*)^3$, für die k Bauern mit ℓ Zügen pro Turmzug mit einer maximalen Zugweite von n den Turm immer fangen können. Trifft dies auf eine Einstellung zu, so trifft es auch auf alle Einstellungen mit größerem n zu.

$$(k,\ \ell,\ n) \in \mathcal{M} \ \Rightarrow\ \forall n'\ (n' \geq n \Rightarrow (k,\ \ell,\ n') \in \mathcal{M})$$

Die zuvor beschriebene Menge $\mathcal{L}$ ist im gewissen Sinne eine Teilmenge mit $n = 1$. Daher sind alle Einstellungen von k und ℓ in $\mathcal{L}$ auch in $\mathcal{M}$ gültig.

$$(k,\ \ell) \in \mathcal{L} \ \Rightarrow\ \forall n\ ((k,\ \ell,\ n) \in \mathcal{M})$$

Die Argumente für $\ell = 1$ halten immernoch. Die für $k = 7,\ \ell = 1$ beschriebene Strategie des Turms basiert nicht auf der Reichweite der Bauern, sondern darauf, dass mit einem Zug nur entweder eine Reihe oder eine Linie besetzt werden kann. Also ist der Turm, unabhängig von n, für alle $k \leq 7 \wedge \ell = 1$ immernoch nicht schlagbar, für alle $k \geq 8$ schon.

$\mathbf{n = 2}$ Das Schlupfloch, dass für $(3,\ 2)$ und $(3,\ 3)$ betrachtet wurde, funktioniert jedoch nicht mehr für $n > 1$, da nun auch ein Bauer zum abdecken einer Bretthälfte aussreicht. Für $k = 3,\ \ell = 2,\ n = 2$ ist der Turm demnach zu fangen. Wenn jedoch $k = 2$, so kann der Turm ein Schlupfloch finden. Für $(2,\ 2,\ 2)$ kann er nicht gefangen werden.

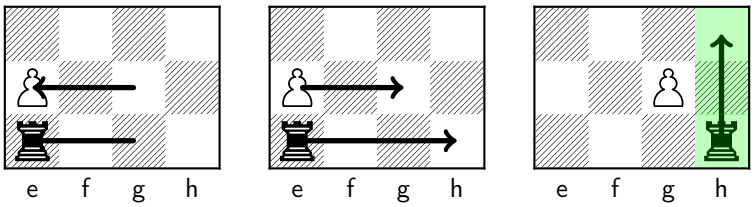

Abbildung 17: Schlupfloch für $(2,\ 2,\ 2)$

n = 3 Unter folgender Konstellation können die Bauern nach jeweiligem Ziehen jede mögliche Kombination aus freier Reihe und Linie besetzen, da der Abstand jeder Reihe und Linie zum nächsten Bauern immer maximal zwei Felder beträgt.

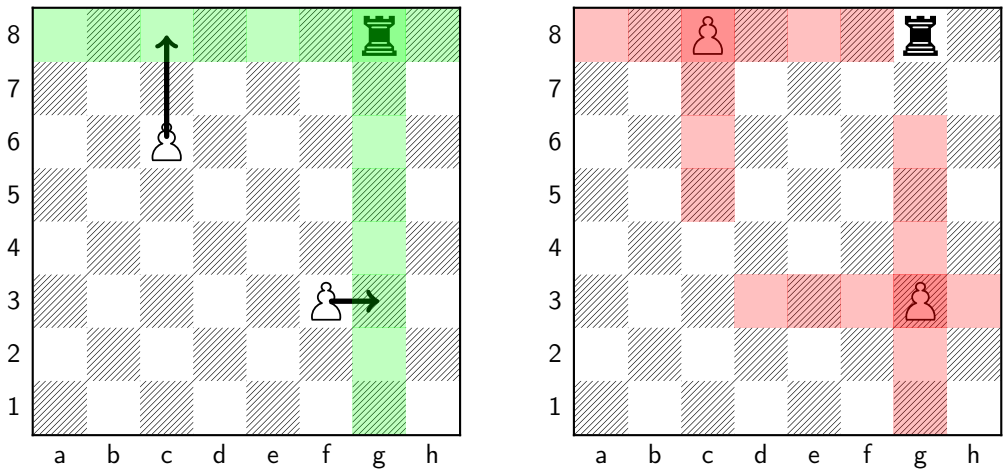

Abbildung 18: Der Turm bekommt ernsthafte Probleme

Während einer der Bauern stets seine Versuche, eine freie Reihe oder Linie zu erreichen, blockiert, wird der Turm vom jeweils anderen immer weiter eingedrängt. Nach kurzer Zeit ist der Turm in die Ecke gedrängt worden und kann geschlagen werden.

Zusammenfassend Mit $n = 2$ ist der Turm auch für $(k = 3,\ \ell = 2)$ und $(k = 3,\ \ell = 3)$ zu fangen. Mit $n = 3$ fällt weiterhin $(2,\ 2)$ weg. Jedoch bleiben, unabhängig von n, die Türme für $(k \leq 7,\ \ell = 1)$ nicht zu fangen. Das bedeutet, dass selbst wenn die Bauern sich selbst wie Türme verhalten dürften $(n = 8)$, dass sie zu siebt mit einem Zug je Turmzug nicht in der Lage wären, ihn zu bezwingen.

Anhang

Ein Optimaler Spielverlauf der Treibjagd

Ta1 Bb6-b5 Ta1-a1 Bd6-d5 Ta1-a1 Bf6-f5 Ta1-a1 Bh6-h5 Ta1-a1 Bc3-c2 Ta1-d1 Bg3-g2 Td1-d1 Bg2-g1 Td1-b1 Bg1-f1 Tb1-b1 Bf1-e1 Tb1-b1 Ba3-a2 Tb1-b3 Bb5-b4 Tb3-b1 Ba2-a1 Bxa1-b1

Benutzung des Programms

Das Programm wurde in `C++11` implementiert. Es wird eine Unix-Umgebung mit `g++` benötigt. Es kann entweder via Konsole oder via grafischer Oberfläche benutzt werden. Dies kann beliebig beim kompilieren gewählt werden. Für eine grafische Darstellung wird die CImg-Library[3] verwendet, was irgenteine Form von Grafiktreiber erfordert.
Für eine GUI muss das Kommando `make gui` verwendet werden, sonst `make` oder `make terminal`. Das Programm ist mit `./chess` auszuführen. Die Eingabe im Terminal erfordert das Feld in üblicher Notation (z. B. `a3`), auf das der Turm als nächstes zieht bzw. das, auf welches er als erstes setzt. Das Programm gibt darauf den Bauernzug zurück. In der Grafikanwendung muss mit der Maus auf das entsprechende Feld geklickt werden. Ungültige Eingaben werden ignoriert und müssen wiederholt werden.

```
$ cd chessim
chessim$ make
g++ -c game.cpp -w -std=c++11 -lm -lpthread -O2
g++ -o chess chess.cpp game.o textInterface.h -w -std=c++11 -lm -lpthread -O2
chessim$ ./chess
a1
ok Bb6-b5
a1-d1
ok Bd6-d5
d1-i9
d1-d2
ok Bf6-f5
...
```

[3] *siehe* `http://cimg.eu/`

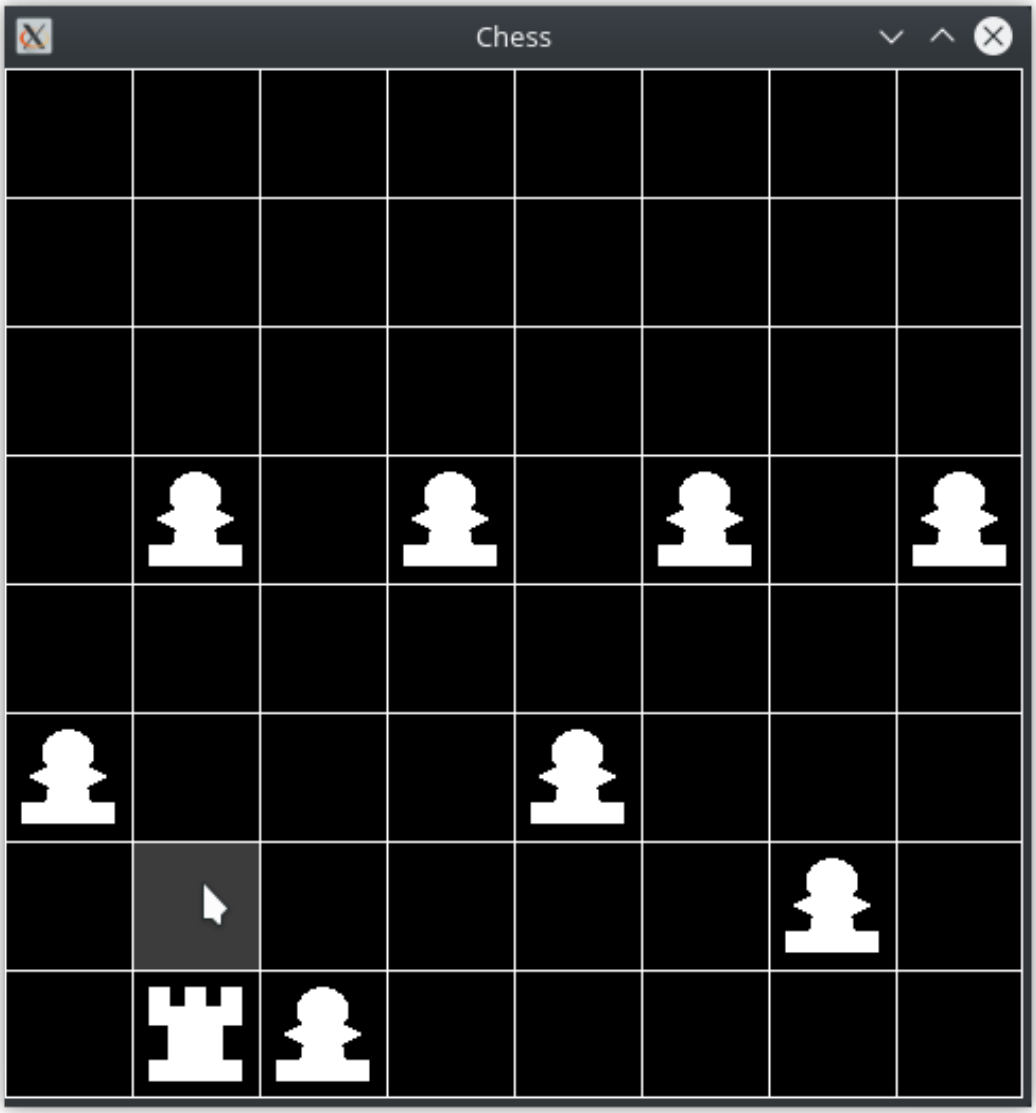

Abbildung 19: Grafische Anwendung

Quellcode

game.cpp

```
/* 36. BWInf Aufgabe 5 - Bauernopfer
   Kamal Abdellatif
*/

#include "game.h"

Tile board[DIM][DIM], *pawns[PAWN_N];
Cord rookX, rookY;
Move bestPawnMove;

void initBoard() {
  /* Initialisierung des Bretts (board) */
  for (Cord x = 0; x < DIM; ++x)
    for (Cord y = 0; y < DIM; ++y)
      board[x][y]= {x, y, EMPTY, false, NULL};
}

void putPawns(const Cord coordinates[PAWN_N][2]) {
  /* Besetzt board mit Bauern aus den Koordinatenpaaren coordinates */
  Cord x, y;
  for (int i = 0; i < PAWN_N; ++i) {
    x = coordinates[i][0];
    y = coordinates[i][1];
    board[x][y].type = PAWN;
    pawns[i] = &board[x][y];
    board[x][y].pawnEntry = &pawns[i];
  }
}

void putRook(const Tile& tile) {
  /* Setzt den Turm auf ein Feld tile */
  rookX = tile.x;
  rookY = tile.y;
}

void clearVisited() {
  /* Setzt den visited-Flag fuer die Tiefensuche zurueck */
  for (Cord x = 0; x < DIM; ++x)
    for (Cord y = 0; y < DIM; ++y)
      board[x][y].visited = false;
}

bool inBounds(Cord x, Cord y) {
  /* Gibt zurueck, ob ein Punkt (x, y) sich auf dem Brett befindet */
  return (0 <= x && x < DIM && 0 <= y && y < DIM);
}

Tile* relativePos(const Tile& start, Dir heading, Cord length) {
  /* Gibt die Position zurueck, welche length Felder von start in
  Richtung heading liegt */
```

```
  Cord x = start.x, y = start.y;
  switch (heading) {
    case RIGHT: x += length;
          break;
    case UP:    y += length;
          break;
    case LEFT:  x -= length;
          break;
    case DOWN:  y -= length;
          break;
  }
  if (!inBounds(x, y)) return NULL;
  return &board[x][y];

}

Tile** nextRookMoves(Tile& start, Tile **buffer) {
  /* Beschreibt **buffer mit Pointern auf alle Felder, die der Turm
  im naechsten Zug erreichen kann */
  bool ended[] = {false, false, false, false};

  Tile *t;
  Tile **next = buffer;
  *(next++) = &start;
  for (Cord length = 1; length < DIM; ++length) {
    for (Dir heading : headings) {
      if (ended[heading])
        continue;
      t = relativePos(start, heading, length);
      if (t == NULL || t->type == PAWN)
        ended[heading] = true;
      else if (isGood(*t))
        *(next++) = t;
    }
  }
  *next = NULL;
  return buffer;
}

void updateMovePossibilities(bool poss[DIM][DIM]) {
  /* Beschreibt den Buffer poss mit Wahrheits werden.
  poss[x][y] == true wenn der Turm das Feld (x, y) in einem Zug
  erreichen kann, sonst false. */
  Tile *buffer[2*DIM];
  for (Cord x = 0; x < DIM; ++x)
    for (Cord y = 0; y < DIM; ++y)
      poss[x][y] = false;

  nextRookMoves(board[rookX][rookY], buffer);
  for (Tile **next = &buffer[0]; *next != NULL; next++) {
    poss[(*next)->x][(*next)->y] = true;
  }
}
```

```
bool isGood(const Tile& tile) {
  /* Gibt zurueck, ob das Feld tile unbedroht ist. */
  if (tile.type == PAWN) return false;
  Tile *neighbor;
  for (Dir heading : headings) {
    neighbor = relativePos(tile, heading, 1);
    if (neighbor == NULL)
      continue;
    if (neighbor->type == PAWN)
      return false;
  }
  return true;
}

bool isGood(Cord x, Cord y) {
  /* Gibt zurueck, ob das Feld (x, y) unbedroht ist. */
  if (!inBounds(x, y)) return false;
  return isGood(board[x][y]);
}

unsigned int countAccessibleTiles(Tile& tile) {
  /* Rekursive Funktion, die die Anzahl der erreichbaren
  Felder bestimmt */
  tile.visited = true;
  unsigned int count = 0;
  if (isGood(tile))
    count ++;
  Tile* buffer[2*DIM];
  nextRookMoves(tile, buffer);
  for (Tile **next = &buffer[0]; *next != NULL; next++)
    if (!(*next)->visited)
      count += countAccessibleTiles(**next);

  return count;
}

unsigned int evaluate() {
  /* Gibt die Anzahl von Zugmoeglichkeiten fuer den Turm zurueck */
  unsigned int count = countAccessibleTiles(board[rookX][rookY]);
  clearVisited();
  return count;
}

void doPawnMove(const Move& move) {
  move.start->type = EMPTY;
  move.end->type = PAWN;
  *(move.start->pawnEntry) = move.end;
  move.end->pawnEntry = move.start->pawnEntry;
  move.start->pawnEntry = NULL;
}

void undoPawnMove(const Move& move) {
  Move reverse = {move.end, move.start};
  doPawnMove(reverse);
```

```
}

void doRookMove(const Move& move) {
  rookX = move.end->x;
  rookY = move.end->y;
}

void undoRookMove(const Move& move) {
  rookX = move.start->x;
  rookY = move.start->y;
}

Move* possiblePawnMoves(Move *buffer) {
  Tile *end;
  Move *move = buffer;
  for (Tile *start : pawns) {
    for (Dir heading : headings) {
      end = relativePos(*start, heading, 1);
      if (end == NULL)
        continue;
      if (end->type == EMPTY) {
        move->start = start;
        move->end = end;
        move++;
      }
    }
  }
  move->start = move->end = NULL;
  return buffer;
}

Move* possibleRookMoves(Move *buffer) {
  Tile *start = &board[rookX][rookY];
  Tile *ends[16];
  Move *move = buffer;
  for (Tile **end = nextRookMoves(*start, ends); *end != NULL; end++) {
    move->start = start;
    move->end = *end;
    move++;
  }
  move->start = move->end = NULL;
  return buffer;
}
```

game.h

```
/* 36. BWInf Aufgabe 5 - Bauernopfer
   Kamal Abdellatif
*/

#ifndef GAME_H
#define GAME_H

#include <stdint.h>
```

```c
#include <stddef.h>

enum Dir {RIGHT, UP, LEFT, DOWN};
enum Type {EMPTY, PAWN};

typedef int8_t Cord;

typedef struct Tile {
  Cord x;
  Cord y;
  Type type;
  bool visited;
  struct Tile **pawnEntry;
} Tile;  // Typ eines Schachfeldes

/*
Wenn ein Feld von einem Bauern besetzt ist, so befindet
sich in *pawns[PAWN_N] ein Pointer auf dieses Feld.
**pawnEntry ist in diesem Fall ein Pointer auf diesen Eintrag.
*/

typedef struct {
  Tile *start;
  Tile *end;
} Move;

const int8_t DIM = 8;  // Dimension des Feldes
const int8_t PAWN_N = 8;  // Anzahl Bauern
const Dir headings[] = {RIGHT, UP, LEFT, DOWN};

extern Tile board[DIM][DIM], *pawns[PAWN_N];
extern Cord rookX, rookY;

void initBoard();
void putPawns(const Cord coords[PAWN_N][2]);
void putRook(Cord x, Cord y);
void clearVisited();

bool inBounds(Cord x, Cord y);
Tile* relativePos(const Tile& start, Dir heading, Cord length);

Move* possiblePawnMoves(Move* buffer);
Tile** nextRookMoves(const Tile &start, Tile **buffer);
void updateMovePossibilities(bool poss[DIM][DIM]);

bool isGood(const Tile& tile);
bool isGood(Cord x, Cord y);
unsigned int countAccessibleTiles(Tile& tile);

void doPawnMove(const Move& move);
void undoPawnMove(const Move& move);
void doRookMove(const Move& move);
void undoRookMove(const Move& move);
```

```
unsigned int evaluate();
unsigned int minPawns(unsigned int depth, unsigned int alpha, unsigned int beta);
unsigned int maxRook(unsigned int depth, unsigned int alpha, unsigned int beta);
Move getBestPawnMove();

#endif
```

chess.cpp

```
/* 36. BWInf Aufgabe 5 - Bauernopfer
   Kamal Abdellatif
*/

#include "game.h"

// Einbinden je nach Terminal oder grafischer Darstellung
#ifdef GUI
# include "draw.h"
#else
# include "textInterface.h"
#endif

unsigned int score, minScore;
Move buffer[32], *move, *minMove;

void react() {
  /* Macht den naechsten Bauernzug nach Strategie */
  possiblePawnMoves(buffer);  // Einlesen aller moeglichen Zuege
  minScore = 10000;

  // Ermittlung des Zugs der die Moeglichkeiten minimiert
  for (move = &buffer[0]; move->start != NULL; move++) {
    doPawnMove(*move);
    score = evaluate();
    if (score < minScore) {
      minScore = score;
      minMove = move;
    }
    undoPawnMove(*move);
  }
  doPawnMove(*minMove);
}

int main(int argc, char const *argv[]) {
  initBoard();
  const Cord coords[][2] = {  // Startstellung
    {0, 2},
    {1, 5},
    {2, 2},
    {3, 5},
    {4, 2},
    {5, 5},
    {6, 2},
    {7, 5},
```

```
  };
  putPawns(coords);  // Setzen der Bauernstellung

  rookX = 0;  // Dummy-Coordinaten des Turms
  rookY = 0;

  bool first = true; // Erster Zug. Ist relevant fuer die Zugmoeglichkeiten.

#ifdef GUI
  do {
    setRookOnClick(first);
    react();
    if (first)
      first = false;
  } while (!userInterrupt());
#else
  while (1) {
    scanRook(first);
    react();
    printMove(*minMove);
    if (first)
      first = false;
  }
#endif
  return 0;
}
```

draw.cpp

```
/* 36. BWInf Aufgabe 5 - Bauernopfer
   Kamal Abdellatif
*/

#include "draw.h"

const unsigned char WHITE[] = {255};

CImg<unsigned char> image(WIDTH, HEIGHT, 1, 1),
                    empty(TILE_SIZE, TILE_SIZE, 1, 1, 0),
                    selected(TILE_SIZE, TILE_SIZE, 1, 1, 60),
                    rook("res/tower.png"),
                    pawn("res/pawn.png");

CImgDisplay disp(image, "Chess", 1, fullscreen);

bool movePoss[DIM][DIM];
clock_t lastClick = clock();

void getMousePos(Cord &x, Cord &y) {
  x = disp.mouse_x()/TILE_SIZE;
  y = DIM - 1 - disp.mouse_y()/TILE_SIZE;
}

void drawGrid() {
```

```
  for (unsigned int i = 0; i < DIM; ++i)
    image.draw_line(TILE_SIZE*i, 0, TILE_SIZE*i, HEIGHT-1, WHITE);
  for (unsigned int i = 0; i < DIM; ++i)
    image.draw_line(0, TILE_SIZE*i, WIDTH-1, TILE_SIZE*i, WHITE);

  image.draw_line(WIDTH-1, 0, WIDTH-1, HEIGHT-1, WHITE);
  image.draw_line(0, HEIGHT-1, WIDTH-1, HEIGHT-1, WHITE);
}

void drawSelected(bool isFirst) {
  Cord x, y; getMousePos(x, y);
  if (!inBounds(x, y))
    return;

  if (movePoss[x][y] || (isFirst && isGood(x, y)))
    image.draw_image(x*TILE_SIZE, (DIM-1-y)*TILE_SIZE, selected);
}

void draw(bool isFirst) {
  for (Cord x = 0; x < DIM; ++x)
    for (Cord y = 0; y < DIM; ++y)
      image.draw_image(
        x*TILE_SIZE, (DIM-1-y)*TILE_SIZE,
        (board[x][y].type == PAWN) ? pawn : empty);

  drawSelected(isFirst);
  if (!isFirst)
    image.draw_image(rookX*TILE_SIZE, (DIM-1-rookY)*TILE_SIZE, rook);
  drawGrid();
  image.display(disp);
}

void setRookOnClick(bool isFirst) {
  updateMovePossibilities(movePoss);
  while (!disp.button()&1 || (float)(clock()-lastClick)/CLOCKS_PER_SEC < COOLDOWN) {
    draw(isFirst);
    disp.wait();
    if (userInterrupt())
      return;
  }  // left mouse button
  lastClick = clock();
  Cord x, y; getMousePos(x, y);
  if (movePoss[x][y] || (isFirst && isGood(x, y))) {  // erstes mal
    rookX = x;
    rookY = y;
  }
  else setRookOnClick(isFirst);
}

bool userInterrupt() {
    return (disp.is_closed() || disp.is_keyESC());
}
```

draw.h

```
/* 36. BWInf Aufgabe 5 - Bauernopfer
   Kamal Abdellatif
*/

#ifndef DRAW_H
#define DRAW_H

#include <time.h>
#include "CImg.h"
#include "game.h"

using namespace cimg_library;

const unsigned int TILE_SIZE = 60;
const unsigned int WIDTH = 8*TILE_SIZE, HEIGHT = 8*TILE_SIZE;
const bool fullscreen = false;

const float COOLDOWN = .1;  // seconds?

void getMousePos(Cord &x, Cord &y);
void drawGrid();
void draw(bool showRook);
void drawSelected();
void setRookOnClick(bool showRook);
bool userInterrupt();

#endif
```

textInterface.h

```
/* 36. BWInf Aufgabe 5 - Bauernopfer
   Kamal Abdellatif
*/

#include <stdio.h>
#include "game.h"

bool movePoss[DIM][DIM];

void scanRook(bool isFirst) {
  /* Liest aus stdin die naechste Position fuer den Turm in rookX, rookY ein */
  updateMovePossibilities(movePoss);
  Cord x, y;
  char file, rank;
  if (!isFirst)
    printf("%c%c-", 'a'+rookX, '1'+rookY);  // ausgeben der aktuellen Position

  scanf(" %c%c", &file, &rank);
  x = file - 'a';  // char operation magic
  y = rank - '1';
  if ((isGood(x, y) && isFirst) || (movePoss[x][y] && !isFirst)) {
    printf("ok ");
```

```
    rookX = x;
    rookY = y;
  } else
    scanRook(isFirst);
}

void printMove(const Move& move) {
  /* Gibt einen Bauernzug move auf stdout aus */
  printf("B%c%c-%c%c\n", 'a'+move.start->x, '1'+move.start->y, 'a'+move.end->x, '1'+move.end->y);
}
```