Seminarska naloga 1

Iskanje šah matov pri 1-2-3 šahu

Danijel Maraž

Fakulteta za Računalništvo in Informatiko UL dm9929@student.uni-lj.si

Abstract. The article covers the work done in the scope of the first seminar assignment as part of the subject Algorithms.

Keywords: A* BFS Progressive Chess Heuristics Pathfinding

1 Uvod

Pri seminarski nalogi nam je bil dan cilj implementiranja algoritma A* (A zvezda) v namen iskanja potez, ki vodijo do šah mata pri dani postavitvi šahovskih figur. Kot jezik implementacije sem si izbral Python, ker menim, da se z njim da pisati lepšo in bolj berljivo kodo. Iz drugega vidika pa je tudi bistveno počasnejši od Jave. Če bi torej program pisal za trg bi ga zagotovo s pomočjo jezika, ki se hitreje izvaja.

2 Struktura programa

```
def BFS_move(brd, c):
    visited = set()
    nextq = [(0, c, id(brd), brd)]
    heapq.heapify(nextq)
    while nextq:

            #Pop best rated board
        current = heapq.heappop(nextq)
        current[3].turn = moveside

            #Add board to visited
        zh = zobrist_hash(current[3])
        visited.add(zh - current[1])

        # Check if the given position is a checkmate
        if current[1] == 0:
            current[3].turn = enemyside
            if current[3].is_checkmate():
                sstr = ""
```

```
            moveslist = [str(m) for m in current[3].move_stack]
            for move in moveslist:
                sstr += move[:2] + "-" + move[2:] + ";"
            print(sstr[:-1])
            break
        current[3].turn = moveside
        continue
```

Funkcija *BFS_move* prejme kot argumenta *brd* (board razred Python šahovkse knjižnice inicializiran iz fen notacije posamezne testne datoteke) in *c* (koliko potez ima program na voljo). Takoj za tem ustvarimo množico *visited*, ki hrani zgoščene slike že videnih šahovnic ter min kopico *nextq*. Ta hrani terke oblike *(ocena, št. potez, identifikator, šahovnica)* in jih med seboj razporeja glede na prvi element. Nato se začne prva while zanka, ki se izvaja vse dokler ne najdemo šah mata ali dokler ni kopica *nextq* prazna. Nadalje iz kopice pridobimo najnižje ocenjeno terko, zgoščeno sliko njene šahovnice dodamo k množici *visited* in ob pogoju, da ta nima več na voljo potez preverimo ali smo prišli do šah mata.

```
# The main loop
for move in current[3].legal_moves:

    sc = current[1]

    current[3].push(move)
    current[3].turn = moveside

    # Eliminate premature attacks on the king
    # Eliminate last moves that don't attack the king
    wic = current[3].was_into_check()
    if (sc > 1 and wic) or (sc == 1 and not wic):
        current[3].pop()
        continue

    # Check if we already saw this board before
    zh = zobrist_hash(current[3])
    if zh - (sc - 1) in visited:
        current[3].pop()
        continue
```

Z zgornjo zanko nato pregledamo zalogo vseh dovoljenih potez in marsikatero že na osnovi osnovnih pravil 1-2-3 šaha izločimo. Potezo porinemo na sklad potez izbrane šahovnice in s pomočjo funkcije *was_into_check()* preverimo ali je bil kralj napaden pred zadnjo potezo in ali smo prišli do zadnje poteze brez, da bi napadli kralja. V obeh primerih se poteza odstrani iz sklada ter zanka vrne na začetek. Potezo prav tako preskočimo, če s pomočjo množice *visited* ugotovimo, da smo šahovnico že obiskali pri danem številu potez.
Bistvene izboljšave na tem področju:

- Najprej smo za vsako novo potezo naredili kopijo šahovnice in na njej izvajali preverjanja kar se je izkazalo kot zelo potratno, saj je kopiranje celotne instance razreda počasna operacija. Boljša alternativa je bila posamezne poteze preprosto poriniti na sklad že obstoječe instance.
- Smiselno bi bilo, da bi takoj preverili ali smo šahovnico že videli, ampak je v praksi operacija zgoščevanja izjemno počasna. Izkazalo se je da je veliko hitrejše poteze najprej izločiti glede na pravila igre kot pa za vsako potezo računati zgoščevalno funkcijo.

```
r = 0
# Reward promotions
prom = move.promotion
if prom is not None:
    r -= -200
    if prom == chess.Piece(5, moveside):
        r -= 300
    if prom == chess.Piece(4, moveside):
        r -= 200

# Desperate measures
if (time.time() - start) > 17:
    rand = random.randint(1, 20)
    r += rand

# Reward last turn
if sc == 1:
    r -= 1000

# Apply various heuristics

# Coverage of squares around the King
r -= KSCoverage(current[3], board.king(enemyside))
# Manhattan distance from the new position of the piece that moved
r += ManhattanLight(move, board.king(enemyside))

# Reward depth
r -= (sc - 1) * 100

ccopy = current[3].copy()
current[3].pop()

heapq.heappush(nextq, (r, sc - 1, id(ccopy), ccopy))
```

Spremenljivka r hrani točke pri ocenjevanju nove šahovnice. Nad njo se izvede vrsta preverjanj, ki tej spremenijo vrednost in na koncu se v min kopico pošlje terka z vsoto vseh ocen.

3 Hevristike

3.1 Promocije

Promocije figur, ki so bistven del igre 1-2-3 šaha se nagrajuje z -200 točkami. Iz izkušenj igranja 1-2-3 šaha vem, da sta najpogostejši figuri pri promocijah kraljica in trdnjava. Zato zanju program prejme dodatnih -300 in -200 točk.

3.2 Naključnost

Če čas izvajanja programa preseže 17 sekund se vsaki novi šahovnici doda naključno število točk med 0 in 20. S tem programu omogočimo večjo dinamičnost pri izbiri naslednje šahovnice in nam naključna srečna izbira lahko drastično izboljša rezultate.

3.3 Nagrajevanje dosega končne poteze

Izkazalo se je, da je najboljše program prisiliti k takojšnjemu pregledu šahovnic, ki nimajo več na voljo potez zato tem prištejemo -1000 točk.

3.4 Pokritost matnega kvadrata

```
def KSCoverage(board, kingpos):
    return 100 * sum([len(board.attackers(moveside, a)) for a in
                      board.attacks(kingpos)
                      if board.is_attacked_by(moveside, a)])
```

Funkcija ugotovi koliko polj matnega kvadrata je že pokritih (napadenih) in za vsako prišteje -100 točk. Pri štetju pokritosti nasprotnikove figure zanemari in dodeli točke za dvojno pokritje polj saj je to bistven del mnogih šah matov. Izkazalo se je, da je ta hevristika najbolj učinkovita, ker zelo lepo opisuje definicijo šah mata. Ob njenem dodatku je program lahko rešil kar 7 dodatnih primerov (Janko, 2015).

3.5 Manhattanska razdalja

```
def Man_dist(sq1, sq2):
    return abs(chess.square_file(sq1) - chess.square_file(sq2)) + abs(
        chess.square_rank(sq1) - chess.square_rank(sq2))


def ManhattanLight(move, kingpos):
    return 100 * (Man_dist(move.to_square, kingpos) - 1)
```

Funkcija za vsako polje Manhattanske razdalje prišteje 100 točk šahovnici. Podobno kot prejšnja hevristika je tudi ta precej nepogrešljiva. Deluje, ker figure prisili k premiku proti matnemu kvadratu. Kljub temu ima svoje pomankljivosti. Predvsem to, da je operacija računanja Manhattanske razdlje od vseh figur do matnega kvadrata precej potratna. Posledično sem se odločil implementirati manj zahtevno verzijo razdalje, ki se izračuna samo za figuro, ki je bila premaknjena in sicer od njene nove pozicije do matnega kvadrata. S tem smo program pohitrili kar nam je pri mnogih primerih prineslo boljše rezultate (Janko, 2015).

3.6 Nagrajevanje globine

Za vsak nivo dosežene globine se oceni prišteje -100 točk. Hevristika je precej uporabna in nam je v mnogih primerih izboljšala rezultate saj program prisili k sledenju boljše ocenjenih globjih šahovnic.

4 Ovrednotev

Rezultati merjenja izboljšav med posameznimi hevristikami		
Brez KingSquares	Brez Manhattan	Vse hevristike
9.788 s	8.104 s	7.208 s
42/60	44/60	46/60

Tabela prikazuje rezultate pri testiranju končne programske rešitve z vsemi hevristikami, rešitev brez upoštevanja Manhattanske razdalje in brez hevristike za ocenjevanje razdalje do kraljevega kvadrata. Za vsako kategorijo sem pet krat pognal testne primere, vsakič računal povprečen čas iskanja rešitve in končnih pet povprečji povprečil v končno povprečje. Opazimo lahko, da je prav hevristika KingSquares najbolj pripomogla k hitrosti in posledično kakovosti rešitve. Teste sem izvajal iz druge Python datoteke brez posebnega testnega okolja pri naslednjih specifikacijah računalnika:

- Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 2.90GHz
- 12.0 GB RAM
- 64-bit Operating System, x64-based processor

5 Programska koda

```
import chess
import sys
import random
import heapq
import time
from chess.polyglot import zobrist_hash

start = time.time()

file = open(sys.argv[1], "r")
fen0 = [line for line in file][0]
fen0s = fen0.split(" ")
avmoves = int(fen0.split(" ")[-1])
startfen = fen0s[-3] + " " + fen0s[-2] + " " + "- - 0 0"
board = chess.Board(startfen)

# Check who plays black or white
color = fen0s[1]
moveside = False
if color == "w":
    moveside = True
enemyside = not moveside


def KSCoverage(board, kingpos):
    return 100 * sum([len(board.attackers(moveside, a)) for a in
                      board.attacks(kingpos)
                      if board.is_attacked_by(moveside, a)])


def Man_dist(sq1, sq2):
    return abs(chess.square_file(sq1) - chess.square_file(sq2)) + abs(
        chess.square_rank(sq1) - chess.square_rank(sq2))


def ManhattanLight(move, kingpos):
    return 100 * (Man_dist(move.to_square, kingpos) - 1)


def BFS_move(brd, c):
    visited = set()
    nextq = [(0, c, id(brd), brd)]
    heapq.heapify(nextq)
```

```
while nextq:

    current = heapq.heappop(nextq)
    current[3].turn = moveside

    zh = zobrist_hash(current[3])
    visited.add(zh - current[1])
    # Check if the given position is a checkmate
    if current[1] == 0:
        current[3].turn = enemyside
        if current[3].is_checkmate():
            sstr = ""
            moveslist = [str(m) for m in current[3].move_stack]
            for move in moveslist:
                sstr += move[:2] + "-" + move[2:] + ";"
            print(sstr[:-1])
            break
        current[3].turn = moveside
        continue

    # The main loop
    for move in current[3].legal_moves:

        sc = current[1]

        current[3].push(move)
        current[3].turn = moveside

        # Eliminate premature attacks on the king + eliminate last moves tha
        wic = current[3].was_into_check()
        if (sc > 1 and wic) or (sc == 1 and not wic):
            current[3].pop()
            continue

        # Check if we already saw this board before
        zh = zobrist_hash(current[3])
        if zh - (sc - 1) in visited:
            current[3].pop()
            continue

        r = 0
        # Reward promotions
        prom = move.promotion
        if prom is not None:
            r -= -200
```

```
            if prom == chess.Piece(5, moveside):
                r -= 300
            if prom == chess.Piece(4, moveside):
                r -= 200

        # Desperate measures
        if (time.time() - start) > 17:
            rand = random.randint(1, 20)
            r += rand

        # Reward last turn attacks on the king
        if sc == 1:
            r -= 1000

        # Apply various heuristics

        # Coverage of squares around the King
        r -= KSCoverage(current[3], board.king(enemyside))
        # Manhattan distance from all non pawn figures to the King
        r += ManhattanLight(move, board.king(enemyside))

        # Reward depth
        r -= (sc - 1) * 100

        ccopy = current[3].copy()
        current[3].pop()

        heapq.heappush(nextq, (r, sc - 1, id(ccopy), ccopy))


BFS_move(board, avmoves)
```

References

Janko, V. (2015). *Razvoj programa za igranje 1-2-3 šaha* (Unpublished doctoral dissertation). Univerza v Ljubljani.