

---

# Protokoll IPC-Projekt

## IPC Game / Pathfinding

---

Softwareentwicklung  
5BHIT 2016/17

Philipp Kogler

Betreuer: Prof. Rafeiner, Prof. Dolezal

GitHub Repository: [1]  
Begonnen am 12. Februar 2017  
Beendet am 12. Februar 2017

# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>1</b>
1.1	Ziele . . . . .	1
1.2	Voraussetzungen . . . . .	1
1.3	Aufgabenstellung . . . . .	1
<b>2</b>	<b>Allgemeine Beschreibung des Algorithmus</b>	<b>3</b>
2.1	Erste Überlegungen und Ziele . . . . .	3
2.2	Definieren des aktuellen Spielmodus . . . . .	3
2.3	Bewerten der Felder . . . . .	4
<b>3</b>	<b>Implementierung des Algorithmus</b>	<b>4</b>
3.1	Vorbereitung / Voraussetzung . . . . .	4
3.1.1	Die Node Klasse . . . . .	4
3.1.2	Die Priority Queue . . . . .	4
3.2	Speichern der Map . . . . .	5
3.2.1	Je nach Feld neue Gewichtung setzen . . . . .	6
3.3	Bewerten der Felder . . . . .	6

# 1 Einführung

Erstelle einen Java- ODER Python-Client, der sich mit dem Server verbindet und selbstständige Entscheidungen trifft!

Dabei soll dieser Client dementsprechend protokolliert und der verwendete bzw. entworfene Algorithmus beschrieben werden.

Zusätzlich soll der Code des Servers analysiert werden und entsprechende Erweiterungen eingebaut werden.

## 1.1 Ziele

Das Ziel der Übung ist, Python und Sockets genauer kennen zu lernen. Somit soll die Inter Prozess Kommunikation nahe gebracht werden.

Zusätzlich dient die Übung als Einblick bzw. Einstieg in das Programmieren einer einfachen KI welche selbst ständig agieren soll

## 1.2 Voraussetzungen

- Grundlagen Python
- Grundlagen IPC
- Grundlagen Pathfinding

## 1.3 Aufgabenstellung

Es soll ein Client, welcher selbstständig Befehle an den Server schickt entworfen werden. Dabei soll darauf geachtet werden, dass dieser den Fallen bzw. den Lakes ausweicht.

Dieser Algorithmus ist entsprechend zu dokumentieren.

Zusätzlich sollen entsprechende Erweiterungen am Server durchgeführt werden.

- Einstellen der Feldgroesse des Servers
- Einstellen der Felder (Mountain, Forest,Lakes)

Bei der Umsetzung des Clients sind folgende Punkte zu beachten.

- Der Client schickt anschließend basierend auf den Antworten des Servers selbstständig (ohne Benutzereingaben) Move-Befehle:
  - ÜP": Nach oben bewegen
  - "RIGHT": Nach rechts bewegen
  - "DOWN": Nach unten bewegen
  - "LEFT": Nach links bewegen
- Der Client lässt die Spielfigur NICHT ins Wasser fallen
- Der Client bewegt sich zur Schriftrolle, wenn er sie sieht
- Der Client bewegt sich zur gegnerischen Burg, nachdem er die Schriftrolle eingesammelt hat
- Ansonsten erkundet der Client die Landschaft - auf der Suche nach Schriftrolle bzw. Burg
- Alle Verbindungen werden sauber geschlossen
- JavaDoc/DocString-Kommentare sowie Dokumentation (Sphinx/JavaDoc) sind vorhanden

## 2 Allgemeine Beschreibung des Algorithmus

### 2.1 Erste Überlegungen und Ziele

Bei dem Entwurf des Algorithmus wurde darauf Wert gelegt den best möglichen Weg zu wählen. Ursprünglich war die Überlegung, alle erhaltenen Felder zu speichern und nach jedem Zug neu zu bewerten. Bei der Bewertung werden sowohl die benötigten Züge (heuristic) zu diesem Node betrachtet, als auch die neuen Felder die dieser Node aufdecken würde.

Nach der Bewertung der einzelnen Nodes, erfolgt die Berechnung bzw. das Erstellen eines optimalen Weges zu diesem Node. Um dies umzusetzen wurde der **A\*** Algorithmus umgesetzt und dementsprechend implementiert. Die Beschreibung des eigentlichen Pathfinding wird im späteren Verlauf des Protokolls erklärt.

Je nach Spielmodus, welcher von der KI definiert wird, berechnen sich neue günstige Felder.

### 2.2 Definieren des aktuellen Spielmodus

Die KI kann sich in 3 verschiedenen Spielmodi befinden. Diese sind wie folgt:

- **Aufdecken von Feldern | bevor Schriftrolle**

Der erste Spielmodus beschreibt die Suche nach der Schriftrolle. Dabei werden möglichst viele Felder aufgedeckt um die Schriftrolle schnell zu finden.

- **Weg zur Schriftrolle**

Nachdem die Schriftrolle gesichtet wurde, muss noch der optimale Weg zu dieser gefunden werden. Dabei muss beachtet werden, dass keine Lakes betreten werden, und dass das Feld keine Ränder hat und somit der direkte Weg nicht immer der schnellste ist.

- **Aufdecken von Feldern | nach Schriftrolle**

Nachdem die Schriftrolle in Besitz ist, muss diese noch zu der gegnerischen Burg gebracht werden. Falls dieses noch nicht aufgedeckt wurde muss ein erneutes Suchen nach unbekannten Feldern durchgeführt werden.

- **Weg zur Burg**

Sobald die gegnerische Burg gesichtet wurde, muss wieder der optimale Weg gefunden werden. Hierbei ist zu beachten, dass der optimale Weg hier durch die geringste Anzahl von Zügen beschrieben wird. Im Gegensatz zum optimalen Weg zur Schriftrolle wo sehr wohl auch Felder aufgedeckt werden sollen um die gegnerische Burg eventuelle früher aufzudecken.

## 2.3 Bewerten der Felder

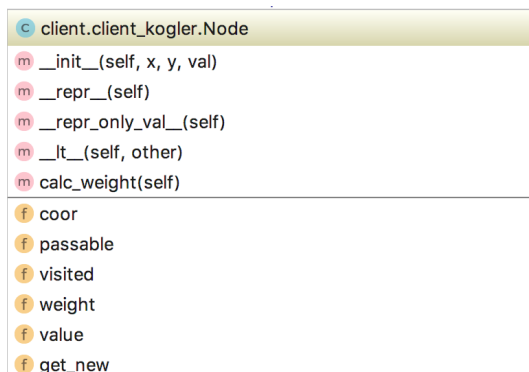
Bei der Bewertung eines Feldes werden mehrere Faktoren betrachtet. Zusätzlich spielt auch der aktuelle Spielmodus eine Rolle. Der wichtigste Faktor ist die möglichen Felder, welche diese Feld aufdecken würde. Zusätzlich wird der Weg bzw. die Anzahl zu diesem Feld in die Berechnung mit einbezogen.

# 3 Implementierung des Algorithmus

## 3.1 Vorbereitung / Voraussetzung

Um den Algorithmus best möglich zu Implementieren wurden gewisse Vorbereitungen getroffen um dies zu erleichtern.

### 3.1.1 Die Node Klasse



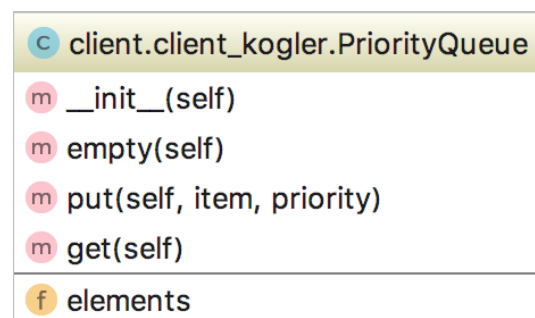
<b>client.client_kogler.Node</b>
<b>m</b> <code>__init__(self, x, y, val)</code>
<b>m</b> <code>__repr__(self)</code>
<b>m</b> <code>__repr_only_val__(self)</code>
<b>m</b> <code>__lt__(self, other)</code>
<b>m</b> <code>calc_weight(self)</code>
<b>f</b> <code>coor</code>
<b>f</b> <code>passable</code>
<b>f</b> <code>visited</code>
<b>f</b> <code>weight</code>
<b>f</b> <code>value</code>
<b>f</b> <code>get_new</code>

Die Node Klasse beschreibt einen Node also ein Feld in der erstellten Map.

Um das Bewerten und das Pathfinding zu erleichtern, hat die Node Klasse bestimmte Methoden. Beispielsweise wird mit `passable` bekannt gegeben ob dieser Node begehbar ist oder nicht. Oder `visited` ob dieser schon besucht wurde.

### 3.1.2 Die Priority Queue

Um bei dem Pathfinding Algorithmus den Nodes eine Gewichtung zu geben oder bei der Berechnung der günstigen Felder in der Nähe eine Priorität zu vergeben, wurde die Priority Queue Klasse entworfen. Mithilfe der Library **heapq** können elemente mit **put** hinzugefügt und mit **get** das Element mit der höchsten Priorität entnommen werden



<b>client.client_kogler.PriorityQueue</b>
<b>m</b> <code>__init__(self)</code>
<b>m</b> <code>empty(self)</code>
<b>m</b> <code>put(self, item, priority)</code>
<b>m</b> <code>get(self)</code>
<b>f</b> <code>elements</code>

## 3.2 Speichern der Map

Bei dem Speichern der erhaltenen Felder, in einer Map dargestellt durch ein 2 Dimensionales Array, wird angenommen dass die eigene Burg gekennzeichnet mit C als Punkt mit den Koordinaten [0,0] behandelt wird.

Um nun alle neu erhaltenen Felder erneut einzuspeichern, muss beachtet werden, wie sich der Spieler im letzten Zug fortbewegt hat und die x bzw. y Position anpassen.

Zusätzlich muss beachtet werden welche Range also Sichtweite das aktuelle Feld liefert.

Die Speicherung wurde wie folgt umgesetzt:

```

1  # Anpassen des letzten Commands
  # x und y demenstreichend veraendern
3  else:
    if prev_command == "up":
5      prev_y -= 1
    elif prev_command == "down":
7      prev_y += 1
    elif prev_command == "left":
9      prev_x -= 1
    elif prev_command == "right":
11     prev_x += 1

13 prev_x = prev_x % size_x
   prev_y = prev_y % size_y
15
   # in der map speichern
17 # for schleife fuer y
   for y in range(0, len(fields)):
19     # for schleife fuer x
       for x in range(0, len(fields)):
21         # temporaerer Node
           node_temp = fields[y][x]
23         # Koordinaten anpassen
           node_temp.coor = ((x-len(fields)//2+prev_x) % size_x, (y-len(fields)//2+prev_y) % size_y)
25         # wenn ein C enthalten ist und dieses nicht an 0, 0 steht
           # als Enemy Castle kennzeichnen
27         if "C" in node_temp.value and node_temp.coor != (0,0):
           node_temp.value = "EC"
29         print("enemy castle found")
           # Node in der Map eintragen
31         map[(y - len(fields)//2+prev_y) % size_x][(x-len(fields)//2+prev_x) % size_y] = node_temp

33 # Kennzeichnen dass das Feld begangen wurde
   map[prev_y][prev_x].visited = True

```

Listing 1: Map Speichern

### 3.2.1 Je nach Feld neue Gewichtung setzen

Nach der erfolgreichen Speicherungen der erhaltenen Felder, müssen diese für den A\* Algorithmus beurteilt werden.

Dafür wird die Gewichtung sowie ob dieser Node begehbar ist, in einem Graphen, welcher von dem A\* Algorithmus verwendet wird, eingetragen

```
1 # graph fuer a* neu beurteilen
2 for y in range(size_y):
3     for x in range(size_x):
4         node = map[y][x]
5         if type(node) == Node:
6             if not node.passable:
7                 # Falls der Node nicht begehbar ist also L
8                 graph.walls.append(node.coor)
9             # Bewerten mit der entsprechenden Gewichtung
10            graph.weights[node.coor] = node.weightv
```

Listing 2: Graph fuer A\* beurteilen

## 3.3 Bewerten der Felder

Nach der berechneten Gewichtung muss das nächste zu besuchende Feld ermittelt werden.

Dafür werden alle Felder erneut bewertet und somit das optimale Feld ermittelt. In die Berechnung wird der benötigte Weg zu diesem Node, als auch die möglichen aufzudeckenden Felder, mit einbezogen.

Die Unterscheidung findet durch den aktuellen Spielmodus statt

Wenn die Bombe noch nicht gefunden wurde werden neue Felder aufgedeckt also wird so bewertet dass das Feld, welches die meisten neuen Felder aufdecken würde ganz oben in der Queue eingereiht wird.

Wenn die Bombe gefunden wurde aber das Gegnerische Castle noch nicht werden wieder Felder aufgedeckt. Falls nun die gegnerische Burg gefunden wurde, wird dieses mit der höchsten Priorität eingereiht.



## Bewerten der Felder nach Modus

```

# map felder bewerten
2 queue = PriorityQueue()
# schleife fuer y
4 for y in range(size_y):
    # schleife fuer x
6     for x in range(size_x):
        node = map[y][x]
8         if type(node) == Node:
            # Nur Nodes die bekannt sind berechnen
            # 0 wird ignoriert
10            if "B" in node.value and have_bomb ==
                False:
12                # bombe gefunden prio ganz oben
                print("bombe gefunden in die queue
                    rein")
14                queue.put(node, -1000000000)
                bomb_node = node.coor
16            elif "EC" in node.value and have_bomb
                == True:
18                # falss bombe schon aufgehoben
                # prio ganz oben
                print("Gehe zu EC")
20                queue.put(node, -1000000000)
            else:
22                # berechnen der aufdeckbaren felder
                # durch das aktuelle feld
24                poss_fields =
                    calc_possible_new_fields(node.
                        value, node.coor, map)
                way_to_field = heuristic(node.coor, (
                    prev_x, prev_y))
26                queue.put(node, (poss_fields * -1) +
                    way_to_field)
                print(poss_fields, "for field:", node)
28                print("way to field:", way_to_field)
                print()
30
# Speichern des Feldes in der Queue
32 next_node = queue.get()
start = (prev_x, prev_y)
34 goal = next_node.coor

```

Listing 3: Felder Bewerten

## Berrechnen der aufdeckbaren Felder

```

def calc_possible_new_fields(val, coor, map):
    my_range = 0
    if "G" in val:
        my_range = 5
    elif "M" in val:
        my_range = 7
    elif "L" in val:
        my_range = 0
    else:
        my_range = 3

    fields_in_range = []

    for y in range(coor[1] - (my_range // 2), coor
        [1] + (my_range // 2 + 1)):
        for x in range(coor[0] - (my_range // 2),
            coor[0] + (my_range // 2 + 1)):
            coor_temp = (x % size_x, y % size_y)
            if coor_temp != coor:
                fields_in_range.append(coor_temp)

    # calculate new fields or not
    for i in fields_in_range:
        for y in range(size_y):
            for x in range(size_x):
                temp = map[y][x]
                if type(temp) == Node:
                    if temp.coor == i:
                        for ii in range(len(
                            fields_in_range)):
                            if fields_in_range[ii]
                                == temp.coor:
                                fields_in_range[ii]
                                    = 0
                        else:
                            pass

    fields_in_range = [x for x in fields_in_range if
        x != 0]

    return len(fields_in_range)

```

Listing 4: Mögliche Felder ermitteln

## Literatur

[1] <https://github.com/pkogler-tgm/Python-IPC-Projekt>.

## Listings

1	Map Speichern . . . . .	5
2	Graph fuer A* beurteilen . . . . .	6
3	Felder Bewerten . . . . .	7
4	Mögliche Felder ermitteln . . . . .	7