Aufgabe 3 - Torkelnde Yamyams

Dominic Meiser

34. BwInf Runde 2

Inhaltsverzeichnis

1 Lösungsidee

Die Aufgabenstellung verlangt, dass man alle sicheren Felder der Karte erkennen soll. Ich habe die Aufgabe umgedreht und zunächst nach allen definitiv unsicheren Feldern gesucht, d.h. Felder, bei denen keine Möglichkeit besteht, dass ein Yamyam von diesem Feld aus einen Ausgang findet. Dazu kann ich einfach von jedem Feld aus alle anderen Felder, die ich erreichen kann, überprüfen, ob diese einen Weg zu einem Ausgang haben. Wenn ich kein solches Feld finde, sondern nur noch Felder, die ich bereits überprüft habe, kann ich sicher sein, dass dieses Feld definitiv unsicher ist. Solche definitiv unsicheren Feldern nenne ich *Failing*. Da ich für jedes Feld im Worst Case alle anderen Felder einmal besuchen muss, hat dies eine Laufzeit $\mathcal{O}(n^2)$, wobei n die Anzahl an Feldern ist, die keine Wand sind.

Nun bleiben zwei weitere Type von Feldern übrig: Die nicht-definitiv unsicheren (im Weiteren als unsicher bezeichnet) und die sicheren Felder. Diese zu bestimmen ist ziemlich einfach und kann mit dem selben Prinzip geschehen wie die *Failing* zu erkennen: Man schaut sich von jedem Feld rekursiv alle Nachbarn an und wenn einer davon als *Failing* markiert wurde, ist das Feld unsicher, und wenn nicht, ist es sicher. Auch dies kann man in $\mathcal{O}(n^2)$ lösen.

1.1 Das Finden von benachbarten Feldern

Die oben als Nachbarn bezeichneten Felder sind aber nicht zwingend die angrenzenden Felder, sondern die Felder, von denen eine (gerichtete) Kante vom Feld zum Nachbar gehen würde, wenn man das Feld als Graphen mit den Knoten als Felder und den Kanten zu den Feldern, in denen das Yamyam das nächste Mal gegen eine Wand stoßen würde und somit seine Richtung ändern würde, auffassen würde. Dementsprechend läuft man einfach so lange in die Richtung, bis man auf eine Wand trifft, und gibt das Feld vor der Wand als Nachbarn an. Eine Außnahme bildet der Fall, in dem vor der Wand ein Ausgang ist. In diesem Fall kann man den Ausgang als Nachbarn ansehen. Dies hat eine lineare Laufzeit, die maximal so groß ist, wie das Feld lang oder breit ist, also $\mathcal{O}(l)$ mit $l = max(width, height))$.

1.2 Die Repräsentation der Welt

Die einzige Funktion, die der oben genannte Algorithmus benötigt, ist ein bestimmtes Feld in der Welt zu finden, und den Typ (Wand, Ausgang oder Leer) und den Status (Failing, Unsicher, Sicher) abzufragen. Dies lässt sich in konstanter Zeit erledigen, indem man die Welt einfach als zweidimensionales Feld repräsentiert.

1.3 Die Eingabedaten

Hierbei kann ich jedoch nicht problemlos dasselbe Eingabeformat wie das von der BwInf-Website benutzen, da ich zu Beginn die Größe der Welt kennen muss. Es ist jedoch kein Problem, die Größe der Welt in die erste Zeile der Datei zu schreiben, um das Array anlegen zu können, bevor man den kompletten Inhalt der Datei eingelesen hat.

1.4 Die Gesamtlaufzeit

Die Gesamtlaufzeit ergibt sich also als $2n$ mal für alle Felder ihre Nachbarn aufrufen (n) und die Nachbarn natürlich auch herausfinden (l). Dementsprechend ist die Gesamtlaufzeit $\mathcal{O}(n^2 + n \cdot l)$ (Die 2 als konstanter Faktor kann vernachlässigt werden).

2 Umsetzung

Ich habe die Lösungsidee in C++11 mit dem Framework Qt (>=5.5) implementiert. Dabei ist es sowohl möglich, den Code mit CMake zu bauen, als auch ihn mit qmake zu bauen.

2.1 Datei `algo.cpp`

2.1.1 Der Algorithmus zur Kategorisierung der Felder

Die Methode `solveWorld` nimmt einen Pointer auf die Welt entgegen und speichert den Status der Felder der Welt in dieser.

```
void solveWorld(World *w)
{
```

Zunächst eine bereits vorhandene Lösung löschen

```
    if (w->hasResult())
    {
        qDebug() << "Warning: Removing existing solution from world";
        for (quint32 i = 0; i < w->width(); i++)
            for (quint32 j = 0; j < w->height(); j++)
                w->field(i, j)->state = World::UnknownState;
    }
```

Den Algorithmus einmal aufrufen und alle Felder, die keinen Ausgang erreichen können, als *Failing* markieren.

```
    for (quint32 i = 0; i < w->width(); i++)
    {
        for (quint32 j = 0; j < w->height(); j++)
        {
            if (w->field(i, j)->type != World::Wall && w->field(i, j)->state == World::UnknownState)
            {
                if (!findPath(w, QPoint(i,j), World::Exit))
                    w->field(i, j)->state = World::Failing;
            }
        }
    }
```

Den Algorithmus nochmal aufrufen und alle Felder, die ein *Failing* Feld erreichen können als *Unsicher* und die restlichen als *Sicher* markieren.

```
    for (quint32 i = 0; i < w->width(); i++)
    {
        for (quint32 j = 0; j < w->height(); j++)
        {
            if (w->field(i, j)->type != World::Wall && w->field(i, j)->state == World::UnknownState)
            {
                if (findPath(w, QPoint(i,j), World::Empty, World::Failing))
                    w->field(i, j)->state = World::Unsave;
                else
                    w->field(i, j)->state = World::Save;
            }
        }
    }

    w->setHasResult(true);
}
```

2.1.2 Der Algorithmus zum Bestimmen des Nachbarn

Die Methode `neighbor` nimmt einen Pointer auf die Welt, den Startpunkt `start` und `xinc` und `yinc` entgegen. Sie geht in jedem Schritt um `xinc`|`yinc` Felder weiter und stopt bei einer Wand oder einem Ausgang.

```
QPoint algo::neighbor(World *w, const QPoint &start, int xinc, int yinc)
{
    Q_ASSERT(qAbs(xinc) <= 1);
    Q_ASSERT(qAbs(yinc) <= 1);
```

Solange weitergehen, bis das Feld nicht *Empty* ist.

```
    int x = start.x(), y = start.y();
    while (w->field(x, y)->type == World::Empty)
    {
        x += xinc;
        y += yinc;
    }
```

Wenn das Feld eine Wand ist, interesiert mich das Feld davor, da das Yamyam nicht in die Wand reinlaufen kann.

```
    if (w->field(x, y)->type == World::Wall)
    {
        x -= xinc;
        y -= yinc;
    }

    return QPoint(x, y);
}
```

2.1.3 Der Algorithmus zum Überprüfen, ob ich ein bestimmtes Feld erreichen kann

Die Methode `findPath` nimmt einen Pointer auf die Welt, den Startpunkt `start`, den zu suchenden Typ bzw. den zu suchenden Status und ein Set mit den bereits besuchten Feldern entgegen und gibt zurück, ob ein solches Feld erreicht werden kann.

```
bool algo::findPath(World *w, const QPoint &start,
                    World::FieldType toFind, World::FieldState state,
                    QSet<QPoint> *searched)
{
    Q_ASSERT(searched);
```

Wenn das aktuelle Feld den Kriterien entspricht, habe ich einen Pfad gefunden

```
    if ((state == World::UnknownState && w->field(start)->type == toFind)
            || (state != World::UnknownState && w->field(start)->state == state))
        return true;
```

Den aktuellen Punkt als besucht markieren

```
    searched->insert(start);
```

Alle Nachbarn finden. Ich möchte hier bemerken, dass ich hierbei jedesmal den Algorithmus aufrufe, um den Nachbarn zu finden, statt dies für jedes Feld nur einmal zu machen und das Ergebnis zu speichern. Da alle Beispiele auf meinem Rechner in unter 10 Millisekunden laufen, war es mir aber den Arbeitsspeicher nicht wert. Die Laufzeit hier ($\mathcal{O}(n^2 + n^2 \cdot l)$) entspricht also nicht der aus der Lösungsidee ($\mathcal{O}(n^2 + n \cdot l)$). Dafür ist die Arbeitsspeicherkomplexität hier konstant, während sie bei der Lösungsidee $\mathcal{O}(n)$ wäre.

```
    QSet<QPoint> neighbors;
    neighbors << neighbor(w, start, -1,  0);
    neighbors << neighbor(w, start,  1,  0);
    neighbors << neighbor(w, start,  0, -1);
    neighbors << neighbor(w, start,  0,  1);
```

Bereits besuchte Nachbarn wieder entfernen. Da in Qt ein Set als Hash Set implementiert ist und neighbors eine maximale Größe von 4 hat, geht dies in konstanter Laufzeit, wie in der Lösungsidee vorgeschrieben.

```
    for (auto it = neighbors.begin(); it != neighbors.end();)
    {
        if (searched->contains(*it))
            it = neighbors.erase(it);
        else
            it++;
    }
```

Rekursiv alle verbleibenden Nachbarn durchsuchen

```
    for (QPoint p : neighbors)
        if (findPath(w, p, toFind, state, searched))
            return true;
    return false;
}
```

2.2 CLI (Command Line Interface)

Das Program wird nur über die Kommandozeile gesteuert. Es gibt dabei folgende Argumente:

build/Torkelnde_Yamyams$ ./yamyams -h

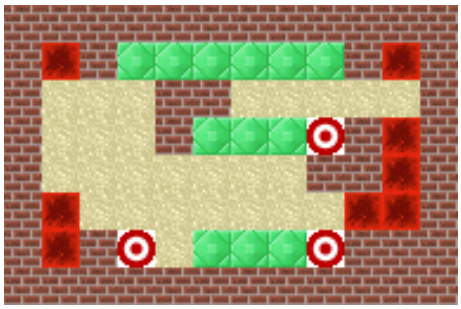
```
Usage: ./yamyams [options]
34. BwInf Aufgabe 3 - Torkelnde Yamyams

Options:
  -h, --help               Displays this help.
  -f, --file <file>        A file containing the description of a world.
  -e, --example <example>  Use the given example from the BwInf.

Elapsed time (in milliseconds): 23.781
```

Wenn -f angegeben wird, werden die Dateien `<file>.orig.yyw`, `<file>.solved.yyw` und `<file>.png` angelegt.

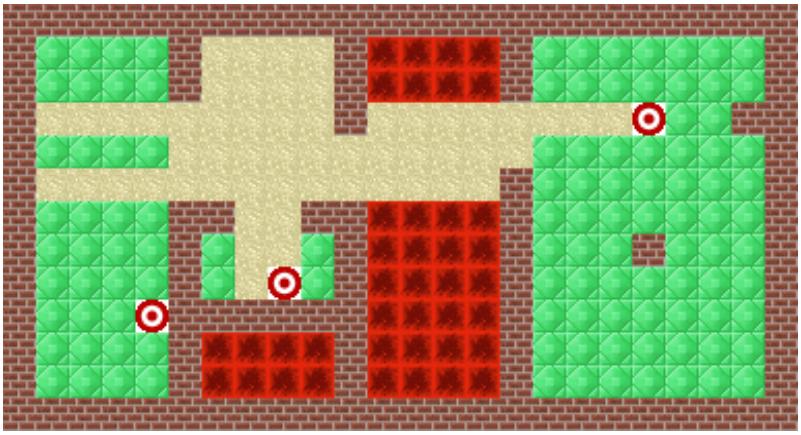
3 Beispiele

In den Beispielbildern ist das rote Feld ein *Failing* Feld, das sandige ein *Unsicheres* und das grüne ein *Sicheres*.

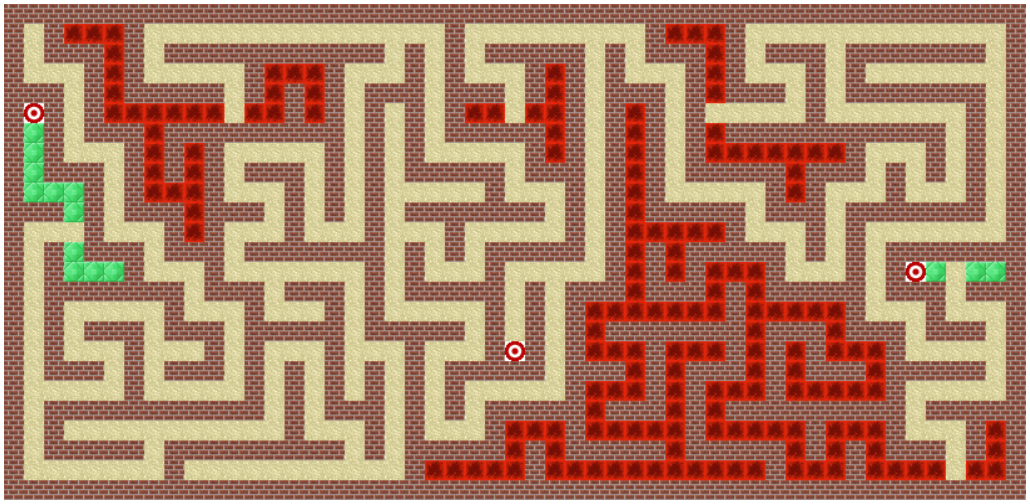
3.1 Beispiel 0 des BwInf



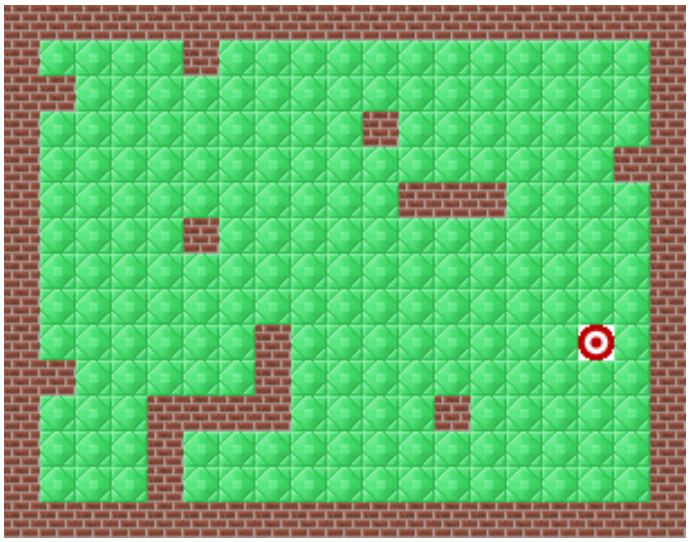
3.2 Beispiel 1 des BwInf



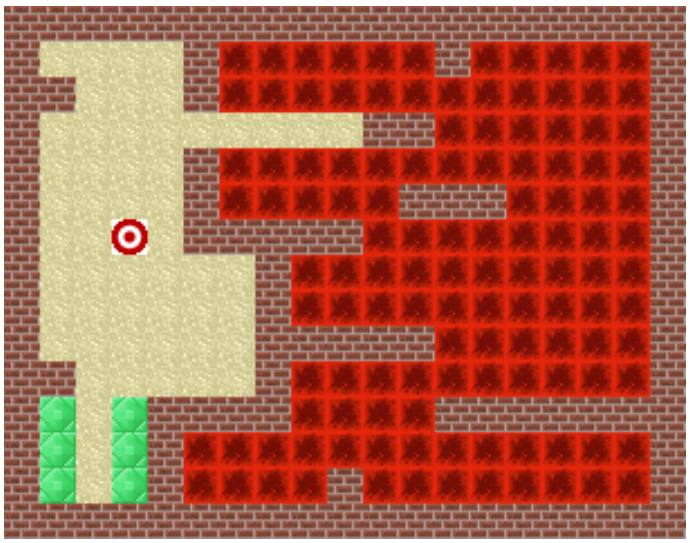
3.3 Beispiel 2 des BwInf



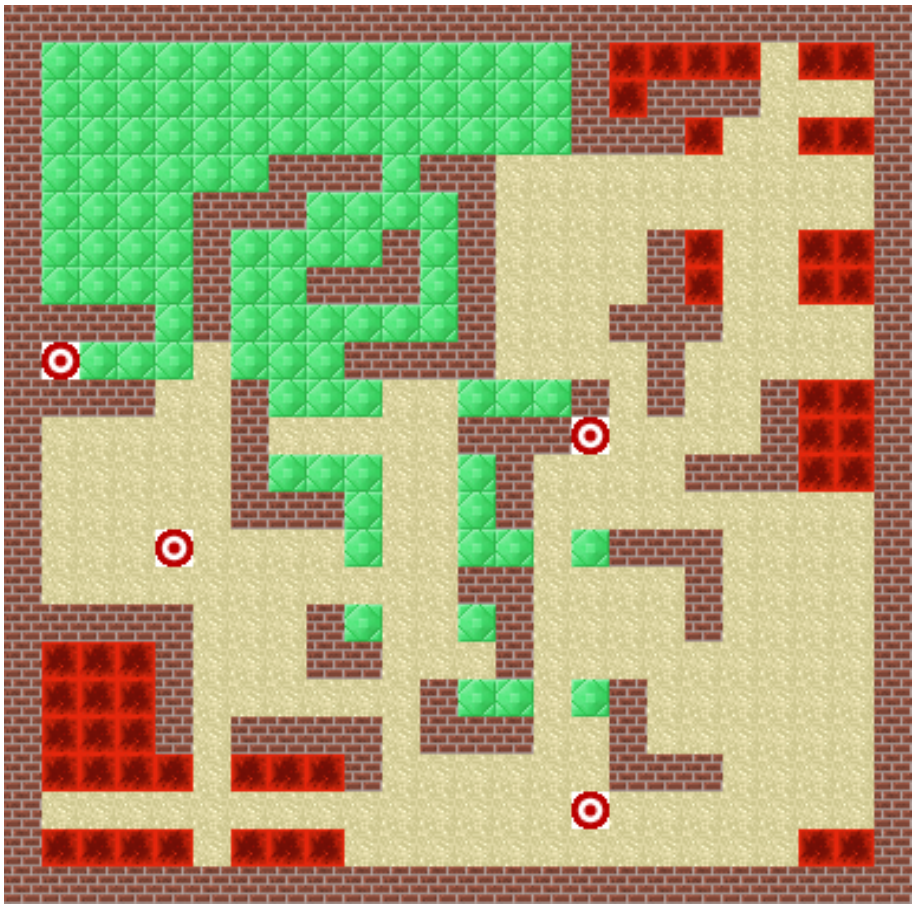
3.4 Beispiel 3 des BwInf



3.5 Beispiel 4 des BwInf



3.6 Beispiel 5 des BwInf



3.7 Beispiel 6 des BwInf