Kassiopeias Weg

Kevin Schier

29. November 2015

1 Lösungsidee

1.1 Erster Teil — Anzahl verbundener Gebiete

Die erste Aufgabe ist es, die Anzahl an verbundenen Bereichen in einem quadratischen Raster zu zählen. Dieses Problem zu lösen, ist auch eine wichtige Optimierung für den zweiten (eigentlichen) Teil der Aufgabe.

Um dieses Problem zu lösen bietet sich ein Algorithmus an wie er in [1] beschrieben wird. In diesem Algorithmus iteriert man reihenweise durch das Raster. Den Feldern werden Bezeichner abhängig von den Feldern links und über dem bearbeitetem gegeben. Wenn das linke Feld den gleichen Inhalt hat, wird dessen Bezeichner übernommen, genauso beim Feld darüber. Wenn das linke und das obere Feld beide den gleichen Inhalt haben wie das überprüfte, wird einer der beiden Bezeichner übernommen und die Bezeichner werden mithilfe der Disjoint-Set Datenstruktur als äquivalent gespeichert. Wenn der Inhalt oben und links unterschiedlich ist, wird ein neuer, bisher nicht genutzter Bezeichner generiert.

In der Disjoint-Set Datenstruktur wird für jede Menge an äquivalenten Elementen ein Element bestimmt, welches das Wurzelelement für diese Menge darstellt. In einem zweiten Durchlauf werden alle Bezeichner im Raster auf dieses Wurzelelement gesetzt. Somit haben alle zusammenhängende Bereiche den gleichen Bezeichner.

Der letzte Schritt ist es die Anzahl verschiedener Bezeichner zu zählen. Dafür sortiert man die Array in der die Bezeichner gespeichert werden, entfernt wiederholte aufeinanderfolgende Elemente und zählt die Anzahl der übrigen Elemente.

Der Algorithmus zur Findung der Bezeichner ist von der Komplexität her asymptotisch $\mathcal{O}(n)$, wobei n die Größe des Rasters ist. Der Sortieralgorithmus am Schluss ist allerdings $\mathcal{O}(n \log n)$.

1.2 Zweiter Teil — Weg, welcher jedes Feld einmal überquert

Unter der Prämisse „Einfach mal losslaufen“ bietet sich hier ein Tiefensuchenalgorithmus an. Man markiert das Feld auf dem Kassiopeia ist als besucht und bewegt sie in eine beliebige Richtung, in der das Feld noch nicht besucht wurde und frei ist. Das wiederholt man solange bis Kassiopeia in einer Sackgasse angekommen ist, wenn noch nicht alle Felder besucht sind, bewegt man sie zurück, entfernt die Markierung und schickt sie in eine andere Richtung.

Auf diese Weise kann die Menge an Wegen sehr schnell explodieren. Auf einem offenen, quadratischen Raster mit 25 Feldern gibt es über 500 Milliarden mögliche Wege zu prüfen. Um

[1] Shapiro, L., und Stockman, G. (2002). Computer Vision. Prentice Hall. S. 69–73

dieses Problem etwas einzudämmen, kann man während der Suche auch wieder die Anzahl an verbundenen Gebieten zählen, wobei man die bereits besuchten Felder auch als nicht passierbar ansieht. Diese Optimierung schränkt die Menge an Wegen enorm ein.

Außerdem kann man die Menge an Sackgassen (Felder mit nur einem angrenzendem freien Feld) zählen, wobei man besuchte Felder wieder als nicht passierbar ansieht. Sobald man mehr als zwei Sackgassen findet, kann man sofort einen Schritt zurück machen, da es nicht mehr möglich sein wird alle Felder zu besuchen, ohne eine besuchtes nochmal zu überqueren.

Mit diesen kleinen Verbesserungen lässt sich für viele Fälle schnell ein Weg finden. Der Algorithmus ist aber immer noch sehr ineffizient in großen, offenen Gebieten in denen kein möglicher Weg existiert.

2 Umsetzung

Die Umsetzung findet in Visual-C++ 2015 statt.

Die Klasse Map enthält alle Daten, die für die Algorithmen benötigt werden. Die Kartendaten werden in einem std :: vector<MapData> gespeichert. MapData ist ein struct aus einzelnen Bits welche speichern, ob ein Feld frei ist, Kassiopeia enthält und schon besucht wurde. Die übrigen fünf Bits in einem Byte werden von den Algorithmen intern genutzt um ein paar Informationen abspeichern zu können.

Das Zählen der verbundenen Gebiete findet in int NumberOfRegions statt. Diese Methode (und alle relevanten anderen) haben Parameter um anzugeben, ob auch markierte Felder oder Kassiopeia als Wand gelten.

Die Suche von Wegen findet in FindFillingPath statt. Diese Methode gibt zurück ob ein Weg gefunden wurde und wenn ja, welcher. Die Kartendaten werden dabei konstant modifiziert, während der rekursiven Suche werden jedoch alle Änderungen auch wieder rückgängig gemacht. So ist es nicht nötig, die Karte mehrmals zu speichern. Die rekursive Suche selbst ist in einer Lambdafunktion innerhalb von FindFillingPath definiert.

Die Disjoint-Set Datenstruktur (disjoint_set) ist grundsätzlich eine Menge an Bäumen. Sie ist auf einer std :: map aufgebaut. Die Schlüssel sind die zu speichernden Elemente und die Werte sind die Elternelemente. Die Wurzel eines Baumes speichert als Elternelement sich selbst. Um die Wurzel eines der Bäume zu finden muss man daher solang den Elternelementen folgen, bis das Elternelement das Element selbst ist. Um zwei Mengen zu vereinen setzt man die Wurzel des einen Baums als Elternelement der Wurzel des anderen Baums. Weitere kleine Optimierungen sind noch mit implementiert[2]. Diese Implementierung wurde aufgrund ihrer Einfachheit ausgewählt, nicht aufgrund der Effizienz. Da Objekte oft mehrmals gespeichert werden, v.A. die Wurzelelemente, werden als Elemente (Bezeichner) einfach nur Zahlen verwendet.

3 Beispiele

Das Beispiel vom Blatt (kassiopeia0.txt). Der Weg wird praktisch ohne Verzögerung gefunden, da die Tiefensuche immernur einen Weg zur Wahl hat.

```
Enter the filename
> kassiopeia0.txt

1: Print Map
2: Show number of regions
```

[2] https://en.wikipedia.org/wiki/Disjoint-set_data_structure#Disjoint-set_forests

```
3: Show filling path
0: Back
> 1
#########
#  #    #
#  # #  #
#  K #  #
#    #  #
#########

1: Print Map
2: Show number of regions
3: Show filling path
0: Back
> 2
Number of regions: 1

1: Print Map
2: Show number of regions
3: Show filling path
0: Back
> 3
Path: WNNWSSSEEENNNEEESSSWNN
```

kassiopeia1.txt; die Suche wird sofort abgebrochen aufgrund des getrennten Gebietes

```
###########
#     #   #
#   ###   #
#   #     #
# K ###   #
#     #   #
###########

Number of regions: 2

1: Print Map
2: Show number of regions
3: Show filling path
0: Back
> 3
Testing:
###########
#     #   #
#   ###   #
#   #     #
# K ###   #
#     #   #
###########
Backtracking...
No path found
```

kassiopeia2.txt; in diesem Beispiel gibt es keinen Pfad, daher findet eine ausgiebige Suche statt, welche im Vergleich zu anderen Fällen relativ lang dauert

```
########
#  K   #
# ###  #
#      #
########
Number of regions: 1
No path found
```

kassiopeia3.txt; nahezu der gleiche Aufbau, wie im letzten Beispiel, allerdings wird hier ein Weg gefunden. Wie so oft bei diesem Algorithmus wird nicht ein einziges mal ein Schritt zurückgegangen, was aber vor Allem am simplen Aufbau des Labyrinths liegt.

```
#########
#  K    #
# ###   #
#       #
#########
Number of regions: 1
Path: EEEESSWNWSWWWWNNE
```

kassiopeia4.txt; hier schlägt die Optimierung zu, Kassiopeia und besuchte Gebiete als Wand beim Zählen der Gebiete anzusehen und die Suche wird sofort abgebrochen.

```
###############
#     K       #
###############
Number of regions: 1
No path found
```

kassiopeia5.txt; keine weiteren Besonderheiten hier

```
###############
#             K#
###############
Number of regions: 1
Path: WWWWWWWWWWW
```

kassiopeia6.txt; hier sind zu viele Sackgassen und die Suche wird sofort abgebrochen

```
#######
#K    #
# # # #
# # # #
#######
Number of regions: 1
No path found
```

kassiopeia7.txt; hier sind wieder zu viele Sackgassen und die Suche wird sofort abgebrochen

```
############
#K  #    # #
# #    #   #
#    #  #  #
############
Number of regions: 1
No path found
```

Ein eigenes Beispiel, um die größte Schwäche des Algorithmus zu verdeutlichen: große, offene Gebiete, in denen kein Weg zu finden ist. Ich habe die Suche bisher nie beenden lassen, da sie unverhältnismäßig lang dauert.

```
####################
#K                 #
#                  #
#                  #
#                  #
#    ###############
#                  #
#                  #
############### # #
#                  #
#                  #
#                  #
####################
Number of regions: 1
This is taking longer than expected - hold on...
```

4 Quelltext

Erzeugung der Bezeichner für die Gebiete.

```
//Two-pass variation stolen from https://en.wikipedia.org/wiki/Connected-component_labeling
std::vector<int> Kassiopeia::Map::PartitionMap(bool considerPathsAsRegions, bool considerTurtleAsWall)
{
  std::vector<int> regionlabels(width * height, -1);
  auto getLabel = [this, &regionlabels](int x, int y)->int& {return regionlabels[y * width + x]; };
   disjoint_set <int> labelset{};
   labelset.insert(-1);

  //1st pass
  {
    auto isEqualColor = [this, &considerPathsAsRegions, &considerTurtleAsWall](int x, int y, const MapData& self)
    {
      return GetMap(x, y).isUsable()
        && (considerPathsAsRegions ? (self.internal_isPath == GetMap(x, y).internal_isPath) : true)
        && (considerTurtleAsWall ? !GetMap(x, y).isTurtle : true);
    };
    int labelcounter = 0;
    for (auto cell : *this)
    {
      if (!GetMap(cell.first.x, cell.first.y).isUsable() || (considerTurtleAsWall&&cell.second.isTurtle)) continue;
      if (cell.first.x > 0 && isEqualColor(cell.first.x - 1, cell.first.y, cell.second))
      {
        if (cell.first.y > 0 && isEqualColor(cell.first.x, cell.first.y - 1, cell.second))
        {
```

```
            getLabel(cell.first.x, cell.first.y) = std::min(getLabel(cell.first.x - 1, cell.first.y), getLabel(cell.first.x,
                cell.first.y - 1));
            labelset.merge(getLabel(cell.first.x - 1, cell.first.y), getLabel(cell.first.x, cell.first.y - 1));
          }
          else
            getLabel(cell.first.x, cell.first.y) = getLabel(cell.first.x - 1, cell.first.y);
        }
        else if (cell.first.y > 0 && isEqualColor(cell.first.x, cell.first.y - 1, cell.second))
        {
          getLabel(cell.first.x, cell.first.y) = getLabel(cell.first.x, cell.first.y - 1);
        }
        else
        {
          getLabel(cell.first.x, cell.first.y) = labelcounter;
          labelset.insert(labelcounter++);
        }
      }
    }

    //2nd pass
    for (auto cell : *this)
    {
      int& label = getLabel(cell.first.x, cell.first.y);
      label = labelset.find(label);
    }

    return regionlabels;
}

int Kassiopeia::Map::NumberOfRegions(bool considerPathsAsRegions, bool considerTurtleAsWall)
{
    std::vector<int> regionlabels = PartitionMap(considerPathsAsRegions, considerTurtleAsWall);
    std::sort(regionlabels.begin(), regionlabels.end());
    return std::distance(regionlabels.begin(), std::unique(regionlabels.begin(), regionlabels.end())) - 1;
}

bool Kassiopeia::Map::IsMapContinuous(bool considerPathsAsRegions, bool considerTurtleAsWall)
{
    return NumberOfRegions(considerPathsAsRegions, considerTurtleAsWall) == 1;
}
```

Erzeugung der Bezeichner für die Gebiete.

```
Kassiopeia::Map::path_result_type Kassiopeia::Map::FindFillingPath()
{
    std::string result;
    result.reserve(width*height);
    for (auto d : *this) d.second.isAlreadyVisited = false;
    ivec2 currentposition = std::find_if(begin(), end(), [this](const MapIterator::value_type& d) {return d.second.isTurtle;
        }).p;
    ivec2 turtleposition = currentposition;
    ivec2 startingturtleposition = turtleposition;

    int printCounter = 0;
    bool printStuff = true;

    auto moveTurtle = [&](ivec2 newPosition)
    {
      GetMap(turtleposition).isTurtle = false;
      turtleposition = newPosition;
      GetMap(turtleposition).isTurtle = true;
    };

    updateAllInternalMarkers();

    std::function<bool(Direction)> recursiveSearch = [&](Direction dir) -> bool
    {
      { //move
        if (dir != Direction::NONE) GetMap(currentposition).isAlreadyVisited = true;
        updateAroundPoint(currentposition);
        currentposition = currentposition + DirectionVectorsWithNone[dir];
```

```
        moveTurtle(currentposition);
      }
      auto rewind = [&]() {
        currentposition = currentposition - DirectionVectorsWithNone[dir];
        GetMap(currentposition).isAlreadyVisited = false;
        updateAroundPoint(currentposition);
        moveTurtle(currentposition);
      };

      if (printCounter++ == 100)
      {
        std::cout << "This is taking longer than expected - hold on..." << std::endl;
        printStuff = false;
      }
      if(printStuff)
      {
        std::cout << "Testing:\n";
        PrintMap();
      }

      //if there is no cell left to be visited, return true
      if (std::find_if(begin(), end(), [](const MapIterator::value_type& d)->bool {
        return d.second.isUsable() && !d.second.isTurtle;
      }) == end())
      {
        rewind();
        return true;
      }

      //skip the search if there are too many dead ends
      std::vector<MapIterator::value_type> deadEnds;
      std::copy_if(begin(), end(), std::back_inserter(deadEnds), [](const MapIterator::value_type& d)->bool {return d.
          second.internal_isDeadEnd; });
      if (deadEnds.size() > 2) goto label_skipSearch;
      else if (deadEnds.size() == 2 && !(deadEnds[0].first == currentposition || deadEnds[1].first == currentposition)) goto
          label_skipSearch;

      //skip search if there is more than one cohesive region
      if (!IsMapContinuous(false, true)) goto label_skipSearch;

      //try to move K in every direction
      for (auto d : Directions)
      {
        ivec2 v = currentposition + DirectionVectorsWithNone[d];
        if (in_range(v, { 0,0 }, { width - 1, height - 1 }) && GetMap(v).isUsable())
        if (recursiveSearch(d))
        {
          result.push_back(DirectionLetters[d]);
          rewind();
          return true;
        }
      }

      label_skipSearch:
      if(printStuff)
      std::cout << "Backtracking...\n";
      rewind();
      return false;
    };

    if (!recursiveSearch(Direction::NONE))
    return path_result_type{ false, std::string{} };

    std::reverse(result.begin(), result.end());

    updateAllInternalMarkers();

    return path_result_type{ true, result };
  }
```