Aufgabe 3
Torkelnde Yamyams

Robert Hönig
Teilnahme-ID: 5776

11. April 2016

Inhaltsverzeichnis

1 Lösung

1.1 Idee

1.1.1 Formalisierung

Zunächst möchte ich die Aufgabenstellung formalisieren. Das Yamyam torkelt in einer Welt, die n Felder breit und m Felder hoch ist. Bezeichnen wir die Welt des Yamyams als Graphen G. Die Felder und Ausgänge dieser Welt sind die Knoten von G, also $V(G)$, wobei $|V(G)| = n * m - |Waende|$. Von jedem Knoten kann man in bis zu vier Richtungen torkeln, also hat jeder Knoten bis zu vier Verbindungen (ich werde von Kanten sprechen) und mindestens null. Liegt in der Welt ein Feld neben einem anderen, so weisen die beiden entsprechenden Knoten jeweils eine eingehende und eine ausgehende Kante auf. Da $|d_G^-(v)|$ und $|d_G^+(v)|$ somit identisch sind, soll $|E(v)| = |d_G^+(v)| = |d_G^-(v)|$ gelten. Liegt ein Feld neben einer Wand, weist das Feld entsprechend keine Kante zu der Wand auf. Da die Menge der ausgehenden Kanten eines Knotens $|E(v)|$ für jede eingehende Kante unterschiedlich ist, können wir keine normalen Graphenalgorithmen auf G laufen lassen. Deswegen unterteilen wir jeden Knoten V in $|E(v)|$ Subknoten, einen für jede eingehende Kante. Sind alle Subknoten eines Knotens *sicher*, so ist auch der Knoten *sicher*. *Sicher* bedeutet dabei, dass man von einem Subknoten keinen *unsicheren* Subknoten erreichen kann. *Unsicher* ist ein Subknoten, wenn es keine Möglichkeit gibt, von ihm einen Ausgang zu erreichen. Nach dem Gesetz der großen Zahlen muss das Yamyam von einem sicheren Knoten also nach $\lim_{t\to\infty}$ einen Ausgang erreichen ($t = Anzahl\,an\,Schritten$).

Die folgende Graphik zeigt die entsprechend untersuchte erste Beispielwelt. Jedes Feld, also jeder Knoten, ist in vier Subknoten aufgeteilt: Zwei, die nach links und rechts, sowie zwei, die nach oben und unten zeigen. Rot sind die unsicheren Subknoten, welche man in den Ecken findet: Hier torkelt das Yamyam lediglich von Ecke zu Ecke, ohne jemals einen Ausgang erreichen zu können. Die Ausgänge sind grün markiert, sichere Subknoten gelb. Orange sind Subknoten, von denen das Yamyam zwar einen Ausgang erreichen könnte, jedoch auch einen unsicheren Subknoten: In diesem Fall würde es auch nie mehr einen Ausgang erreichen, der Subknoten wäre also nicht komplett sicher.

1.1.2 Heuristische Herangehensweise

Kommt man zur Entwicklung eines Algorithmus, ist intuitiv die heuristische Herangehensweise die naheliegendste: Um herauszufinden, von welchen Subknoten aus man definitiv einen Ausgangsknoten erreicht, starte man einen Suchprozess an einem beliebigen Knoten. Anschließend bewege man sich weiter in eine zufällige Richtung. Gelangt man zu einem Ausgang, so erhöht sich die Wahrscheinlichkeit für alle vorher besuchten Subknoten, dass sie sicher sind. Je länger man jedoch auf keinen Ausgang trifft, desto stärker sinkt die Wahrscheinlichkeit für alle vorher besuchten Subknoten, dass sie sicher sind. Damit man nicht in eine Endlosschleife gerät, wenn man auf einen unsicheren

Beispielgraph 1 - gelöst

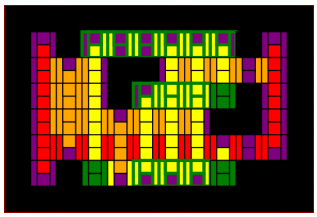


Abbildung 1: grün - Ausgang, gelb - sicher, orange - fraglich, rot - unsicher, lila - an Wand, nicht relevant, grün umrandet - sicheres Feld

Subknoten trifft, sollte der Suchprozess nach einer definierten Anzahl an Knotenbewegungen N_b von einem neuen Subknoten neu gestartet werden. Schließlich determiniert der Algorithmus nach einer definierten Anzahl N_s an Suchprozessen. Die danach für jeden Subknoten berechnete Wahrscheinlichkeit gibt an, wie wahrscheinlich dieser Subknoten sicher, bzw. unsicher ist.

Komplexität Dieser randomisierte Algorithmus der Klasse BPP hat den Vorteil, dass man ihn komplett an die verfügbare Rechenzeit anpassen kann. Seine Zeitkomplexität liegt bei $O(N_b * N_s)$, da er für jeden Suchprozess bis zu N_b Subknoten durchläuft.

Das Ergebnis wird je genauer, je länger man ihn laufen lässt. Zudem benötigt auch nur $O(N_b)$ Speicherplatz, da das die maximale Anzahl an Subknoten ist, die im Speicher gehalten werden muss. Der Nachteil dieses Algorithmus ist, dass er nicht deterministisch ist. Der Benutzer erhält ja schließlich nur Wahrscheinlichkeiten für jeden Knoten und kann sich nicht sicher sein, ob der Knoten nun tatsächlich sicher bzw. unsicher ist.

1.1.3 Deterministische Herangehensweise

Aus diesem Grund möchte ich noch einen zweiten, deterministisch polynomiellen Algorithmus erläutern. Wie bereits erklärt, kann G einige unsichere Subknoten enthalten, also Subknoten, von denen aus man keinen Ausgang erreichen kann. Das können z.B. Eckknoten sein, die in einem halboffenem Rechteck eingeschlossen sind, wie in Abbildung 1 die rechten Subknoten. Gelangt das Yamyam z.B. auf diesen Knoten, springt es zwangsläufig zwischen den Ecken hin und her. Des Weiteren kann man von allen Subknoten, die nicht unsicher sind, mit einer Wahrscheinlichkeit von $P(Subknoten_{fraglich}) > 0$ einen Ausgang erreichen. Kann man von diesen fraglichen Subknoten nun noch nie einen unsicheren Knoten erreichen, gilt wieder nach dem Gesetz der großen Zahlen, dass der fragliche Knoten sicher sein muss: Schließlich wird man von ihm aus nie einen Subknoten erreichen, von dem aus man keinen Ausgang mehr erreichen kann. Der Algorithmus sieht nun also so aus:

Beispielgraph 1 - unsicherer Bereich

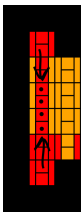


Abbildung 2: Das Yamyam kann nur hin- und herlaufen.

1. Finde alle unsicheren Subknoten
   (a) Teste durch eine Rückwärts-Tiefensuche (rDFS), beginnend von jedem Ausgang, von welchen Subknoten aus man diesen Ausgang erreichen kann. Markiere diese Subknoten. Alle am Ende nicht markierten Subknoten können nie einen Ausgang erreichen, sie sind unsicher.
2. Finde alle sicheren Subknoten
   (a) Teste durch eine rDFS, beginnend von jedem unsicheren Subknoten, von welchen Subnoten aus man diesen ihn erreichen kann. Markiere diese Subknoten. Da sie theoretisch einen unsicheren Subknoten erreichen können, von dem aus man nie mehr einen Ausgang erreichen kann, sind sie nicht sicher.

Für beide Teile braucht man drei Matrizen: Eine, die für jeden Subknoten von G speichert($\mathbf{S}^\intercal$), ob er sicher, unsicher ist (oder ob das noch nicht bekannt ist), sowie eine, die speichert, ob die Tiefensuche ihn schon besucht hat($\mathbf{B}^\intercal$). Das sind die Zustandsmatrizen. Außerdem brauchen wir noch eine, die den Status der Welt speichert, also für jeden Knoten (Feld) die Information liefern kann, welchen Typ die an ihn angrenzenden Felder haben (Ausgänge, Felder oder Wände) ($\mathbf{AdjM}^\intercal$). Eine graphische Darstellung der Speicherstruktur findet sich in Abbildung 3. Die Struktur von $\mathbf{AdjM}^\intercal$ ist identisch, jedoch enthält sie statt Vektoren nur Integer in dem Vektor, hier gibt es keine Subknoten. Durch die Matrixstruktur kann man den Typ aller angrenzenden Felder des Feldes (x, y) bestimmen, indem man die Felder $(x-1, y)$, $(x+1, y)$, $(x, y-1)$ und $(x, y+1)$ überprüft. Für beide Teile führen wir eine rDFS durch. Bei einer Tiefensuche würden rekursiv alle Subknoten aufrufen, die man von einem anderen Subknoten $V(s)$ aus erreichen kann. Bei der rDFS hingegen rufen wir rekursiv alle Subknoten auf, von denen aus man $V(s)$ erreichen kann. Die Adjazenz zur Bestimmung dieser Subknoten ist hier durch $\mathbf{AdjM}^\intercal$ gegeben. Eine graphische Darstellung der Bestimmung dieser Subknoten findet sich in Abbildung 4. Währenddessen setzt man für jeden besuchten Subknoten den Status in ($\mathbf{B}^\intercal$ auf *besucht*, damit er nicht noch einmal besucht wird. Dieser $besucht - Status$ bleibt über alle rDFS erhalten. Sobald nämlich die rDFS alle Subknoten besucht hat, die den Äusgangsausgang"$V(s)$ erreicht haben, gelten

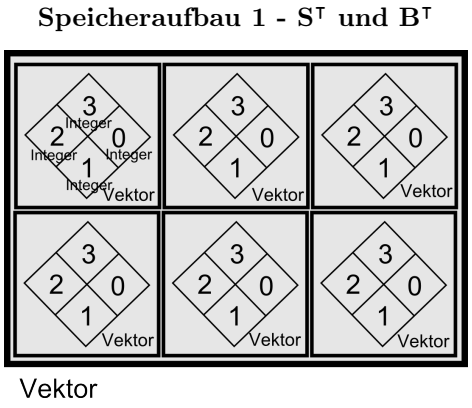


Abbildung 3: Die Speicherstruktur der Zustandsmatrizen - Die Nummern geben den entsprechenden Index im Vektor an, das Karofeld zeigt in die Richtung und enthält den Integer für den Zustand

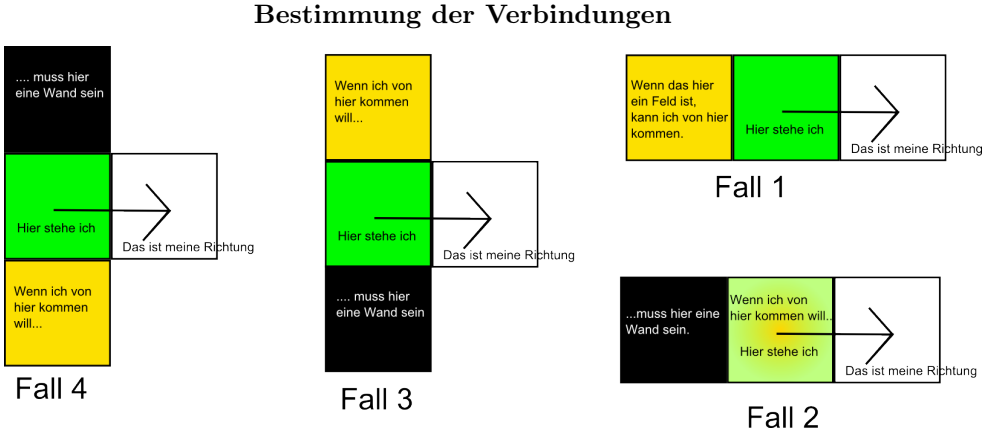


Abbildung 4: Für jede der insgesamt 4 verschiedenen möglichen Verbindungen gibt es Bedingungen, die erfüllt sein müssen, damit das Yamyam von dort auf diesen Subknoten torkeln kann.

diese ja bereits als nicht unsicher. Wenn dann die rDFS vom nächsten Ausgang durchgeführt wird, wäre es unnötig, diese Subknoten noch einmal zu besuchen. Nach Teil 1 wird $\mathbf{B}^\intercal$ zurückgesetzt, damit Teil 2 wieder alle Knoten zur Verfügung stehen. Vorher aber wird jeder Subknoten in $\mathbf{B}^\intercal$ überprüft, ob er besucht wurde. Falls nein, wird er dem Vektor $unsicherN$ hinzugefügt. Diese unsicheren Knoten bilden dann die Basis für Teil 2.

Für Teil 2 können wir genau den gleichen Algorithmus anwenden, mit einem Unterschied: Zweitens prüfen wir in dieser rDFS nicht, welche Subknoten einen Ausgang erreichen können, sondern welche Subknoten einen unsicheren Subknoten erreichen können. Dazu führen wir rDFS wie in Teil 1 aus. Die Startsubknoten $V(s)$ bilden jedoch nicht die Ausgänge, sondern die unsicheren Subknoten $unsicherN$. Die dann am Ende in $\mathbf{B}^\intercal$ markierten Subknoten sind alle fraglich, es sei denn, sie wurden vorher schon als unsicher markiert. Alle jetzt unmarkierten Subknoten, die nicht unsicher sind, sind aber sicher. Sind jetzt alle Subknoten eines Knotens sicher, ist auch der Knoten sicher. Somit haben wir das Ziel erreicht und alle sicheren Knoten bestimmt. Als Nebenprodukt und Zusatz haben wir auch alle unsicheren und fraglichen Knoten bestimmt.

Komplexität Zur Laufzeit dieses Algorithmus lässt sich sagen, das in jedem Teil bis zu $|V(G)|$ Mal eine rDFS gestartet wird: Im ersten Teil, wenn alle Knoten Ausgänge sind. Im zweiten Teil, wenn alle Knoten unsicher sind. Nun ruft die rDFS aber nur Knoten auf, die nicht noch nicht in anderen rDFS besucht wurden. Deswegen werden in jedem Teil maximal $|V_{sub}(G)|$ Subnoten aufgerufen. Jeder Aufruf eines Subknotens verläuft in konstanter Zeit, da er nur den Status des Subknotens zu besucht setzt, und eventuell noch weitere Subknoten aufruft, wobei er ein konstantes Maximum von 4 Verbindungen aufweist, von denen er erreicht werden kann (eine in jede Richtung). Realisiert man ($\mathbf{AdjM}^\intercal$) mit Vektoren, lassen sich diese Verbindungen in konstanter Zeit herausfinden. Somit ist die Laufzeit eines Aufrufs eines Subknotens gleich $O(1)$. Daher ist die Laufzeit der rDFS gleich ($O(n)$, mit n $= |V_{sub}(G)|$) und die Lauzeit des Algorithmus ($O(2 * n) = O(n)$, womit er in der Klasse P liegt. Der Algorithmus läuft in einer Zeit linear zu der Anzahl an Knoten (jeder Knoten hat ja maximal 4 Subknoten).
An Speicherplatz benötigt der Algorithmus drei Vektoren der Größe $|V_{sub}(G)|$, die die bereits erklärten Status speichern müssen. Der Vektor ($\mathbf{AdjM}^\intercal$) speichert den Typ jedes Knotens, womit sein Speicherbedarf bei $|V(G)|$ liegt. Deswegen liegt die Platzkomplexität des Algorithmus mit $O(n)$ in PSPACE, wobei n die Anzahl der Knoten bezeichnet.

1.1.4 Entscheidung

Da der zweite Algorithmus immer richtige Ergebnisse produziert und mit einer linearen Laufzeit alle Beispieleingaben des BWINF, die ja alle höchstens einige tausend Knoten haben, schnell lösen kann, entscheide ich mich für den zweiten,

deterministischen Algorithmus. Sein linearer Speicherplatzbedarf ist dabei auch völlig vertretbar, in der Praxis beläuft er sich auf wenige Megabyte.

1.1.5 Verallgemeinerung und Erweiterung

Die Bedingungen für die gegebene Welt sind sehr begrenzt. Der gegebene Algorithmus lässt sich jedoch im Prinzip auch auf beliebige gerichtete Graphen anwenden. Konkret bedeutet dass, das man die Anzahl an Kanten eines Knoten beliebig variieren kann. Außerdem will ich von den starren Regeln zum Torkeln des Yamyams abweichen: statt eine feste Kante zu haben, die man begeht wenn man den Knoten erreicht, und alle Kanten besuchen kann, sollte man eine Wand treffen, gibt es keine Wände mehr. Stattdessen besitzt ein Knoten für jeden Eingang eine Liste an Ausgängen, zu denen er weitertorkeln kann. Damit kann man nicht nur jede erdenkliche Yamyam-Welt realisieren, sondern auch z.B. Graphen, in denen ein Knoten nur Eingänge, aber keine Ausgänge hat. Das Prinzip der Subknoten gilt auch hier; jeder Eingang zu einem Knoten stellt einen seiner Subknoten dar, der zu einem anderen Knoten führt. In der folgenden Beispielgraphik wurden sie jedoch der Einfachheit halber in die Knoten integriert, eine Liste neben jedes Eingangs eines Knoten zeigt die von dort aus erreichbaren Ausgänge an, wenn man von diesem Knoten kommt.

Beispielgraph 1 - gelöst

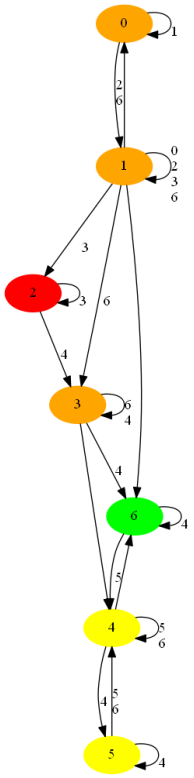


Abbildung 5: grün - Ausgang, gelb - sicher, orange - fraglich, rot - unsicher

Abbildung 5 zeigt einen Beispielgraphen, nachdem er evaluiert wurde. Der Knoten 1 hat einen eingehenden und drei ausgehende Kanten, sowie eine, die auf sich selbst gerichtet ist. Die ausgehende Kante zu Knoten 2 sagt aus, das ich von Knoten 1 den Knoten 2 erreichen kann, und danach aber nur auf Knoten 3 gehen kann. Ginge ich zu Knoten 2 und dann entsprechend 3, kann ich danach nur zu Knoten 4 gehen. Da an der Kante zu Knoten 4 keine möglichen weiteren Verbindungen stehen, kann ich meine Reise bei Knoten 4 nicht fortsetzen, womit der Knoten den Ausgang 6 nicht erreichen könnte. Deswegen kann Knoten 1 nicht sicher sein. Sollte ich von Knoten 1 zu Knoten 3 gehen, kann ich danach meine Torkelei nur nach Knoten 6 fortsetzen. In diesem Fall hätte ich einen Ausgang erreicht, womit Knoten 1 definitiv nicht mehr unsicher sein kann. Die Schleife des Knoten 1 besagt, dass wenn ich bei Knoten 1 starte, ich zu Knoten 0, 2, 3 und 6 torkeln kann. Dadurch muss eine Überprüfung des Knoten 1 auf Sicherheit all diese ausgehenden Kanten berücksichtigen, die ja sowohl sicher als auch unsicher sind. Deswegen ist Knoten 1 fraglich. Äquivalent kann man die anderen Knoten überprüfen.

Zur Berechnung der Knotenstatus kann man denselben Algorithmus wie bei der Rechteck-Welt verwenden:

1. Finde alle unsicheren Subknoten
   (a) Teste durch eine Rückwärts-Tiefensuche (rDFS), beginnend von jedem Ausgang, von welchen Subknoten aus man diesen Ausgang erreichen kann. Markiere diese Subknoten. Alle am Ende nicht markierten Subknoten können nie einen Ausgang erreichen, sie sind unsicher.
2. Finde alle sicheren Subknoten
   (a) Teste durch eine rDFS, beginnend von jedem unsicheren Subknoten, von welchen Subnoten aus man diesen ihn erreichen kann. Markiere diese Subknoten. Da sie theoretisch einen unsicheren Subknoten erreichen können, von dem aus man nie mehr einen Ausgang erreichen kann, sind sie nicht sicher.

Gingen wir von denselben Speicherstrukturen aus, wäre auch die Zeitkomplexität dieselbe: Die Zustandsmatrizen $\mathbf{S^T}$ und $\mathbf{B^T}$ speichern dieselben Zustände wie vorhin. Da diesmal die Adjazenz jedoch nicht mehr durch eine Regel definiert ist (ein Knoten kann theorethisch alle anderen Knoten erreichen, nicht mehr nur maximal 4), stehen wir vor einem Problem: Die Matrix $\mathbf{AdjM^T}$ müsste dann, um für jeden beliebigen Knoten $V(n)$ die Liste der Subknoten $VDavor$, von den man kommen kann, um $V(n)$ zu erreichen, angeben zu können, für jeden Knoten $V(n)$ einen Vektor der Größe $|V(G)|$ enthalten, von der dann wiederum jeder erreichbare Knoten einen Vektor $VDavor$ enthält. Dasselbe gilt auch für die Zustandsmatrizen. Die Platzkomplexität läge also bei $O(|V(G)|^2 + N)$, wobei N die Anzahl der Knoten in allen Vektoren $VDavor$ angibt (siehe Abbildung 6, tiefste Ebene). N ist dabei maximal $|V(G)|^3$: Dafür müsste jeder Knoten mit jedem verbunden sein und $VDavor_e, e \in E(G)$ für alle e alle Knoten enthalten. Wir können den $O(|V(G)|^2)$-Teil für alle Matrizen auf $O(|V(G)| + N)$ verringern. Dazu speichert eine Matrix nur noch die tatsächliche Anzahl der $|V(G)|$ möglichen Verbindungspartner in einer Map $MVerb$ (diese speichert also die Kanten). Diese Map enthält als Key die Nummer des Knotens und als Value $VDavor$, bzw. den Status des Knotens für die Zustandsmatrizen. Im Gegensatz zum vorherigen Algorithmus kann dieser Algorithmus die Verbindungen direkt aus $\mathbf{AdjM^T}$ ablesen. Befinden wir uns bei Knoten $V(s1)$, greifen wir auf diesen in $\mathbf{AdjM^T}$ zu, wir geben ihn als Index für den obersten Vektor an. In der daruenterliegenden Map geben wir den Knoten $V(s2)$ an, der die rDFS für $V(s1)$ gestartet hat, also der Knoten, zu dem wir kommen könnten. Der als Value in dieser Map enthaltene Vektor $VDavor$ enthält dann alle Knoten, von denen man zu $V(s1)$ gehen kann und danach noch $V(s2)$ erreichen kann. Alles Knoten in $VDavor$ rufen wir rDFS auf. Dabei übergeben wir als Parameter $V(s1)$, damit rDFS weiß, von wo man kommt. Erreichen wir in der Tiefensuche Knoten n, müssen wir wissen, woher wir gekommen sind.

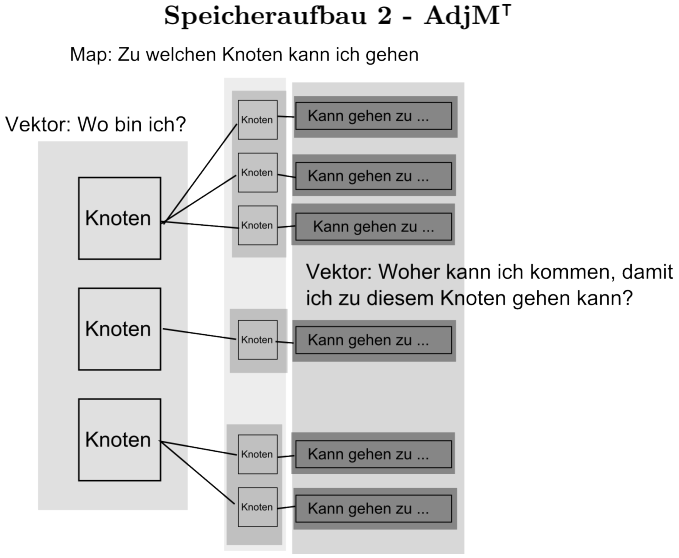


Abbildung 6: Die Speicherstruktur von **AdjM$^\intercal$**. für die Zustandsmatrizen ersetzt man einfach die letzten Vektoren durch einen Integer, der den Zustand angibt.

Für jeden der maximal $|V(G)|$ aufgerufenen Knoten in rDFS wird der entsprechende Vektor $VDavor$ durchlaufen, der eine maximale Größe von $|V(G)|$ hat: Die Laufzeit beliefe sich auf $O(|V(G)|^2)$. Die Durchsuchung der Map nach $V(s2)$ eine Laufzeit von $O(log(|V(G)|))$ hat. Diese Durchsuchung wird maximal für Knoten in rDFS einmal durchgeführt, somit läge die Laufzeit bei $O(|V(G)| * log(|V(G)|) + |V(G)|^2) = O(|V(G)|^2)$. Das gilt jedoch nur bei sehr dichten Graphen. Bei Graphen mit wenigen Kanten tendiert die Laufzeit wird $VDavor$ kleiner, die Laufzeit tendiert zu $O(|V(G)| * log(|V(G)|) + |V(G)|) = O(|V(G)| * log(|V(G)|))$.

1.1.6 Schwächen

Während ich es mich traue, den ersten linearen Algorithmus als optimal zu bezeichnen, kann man das nicht von der generalisierten Methode behaupten. Durch ein komplizierteres System aus Pointern statt einer Knotennummerierung, die Knoten identifizieren, wäre es möglich, die Map mit ihrer logarithmischen Suchzeit zu ersetzen. Die Poiner, die z.B. in einem Vektor, auf den man mit den Knotenzahlen als Indizes zugreifen kann, gespeichert werden können, würden direkt zum gewünschten Vektor $VDavor$ zeigen. Bei Graphen mit wenigen Kanten kann das einen Unterschied in der Laufzeit darstellen.

1.2 Implementierung

Die Implementierung besteht aus insgesamt 5 Programmen bzw. Tools, die in C++11 geschrieben wurden. Die Hauptprogramme sind "Torkelnde Yamyams“ und "Torkelnde Yamyams Abstrakt". Ersteres löst die Aufgabenstellung mit dem deterministischen Algorithmus, letzters stellt die Implementierung des verallgemeinerten Algorithmus dar. "Torkelnde Yamyams"benutzt ein anders Eingabeformat als die Beispiele des BWINF. Das Tool "YamyamBWINFToNormal"kann die Eingaben des BWINFs in mein Format konvertieren. Das Eingabeformat für "Torkelnde Yamyams Abstrakt“ist wieder anders. Da hier die Erstellung von Testfällen jedoch recht kompliziert und fehleranfällig ist, habe ich das Tool "Yamyam Konvertierer"geschrieben. Es nimmt eine Eingabe des normalen Eingabeformats und konvertiert sie in das “AbstraktFormat. Das geht, da sich jede Rechteck-Yamyam-Welt auch in einen entsprechenden Graphen transformieren lässt. Als letztes gibt es noch das Programm "YamYam Graphik", welches die Ergebnis-Datei von "Torkelnde Yamyams“in eine graphische SVG-Datei aufarbeitet. "Torkelnde Yamyams Abstrakt"hingegen erzeugt eine GV-Datei in der DOT-Sprache als Ergebnis, welche sich mit dem kostenlos verfügbaren Tool dot in ein Bild aufbereiten lässt.

1.2.1 Konkretes Programm ”Torkelnde Yamyams”

Zunächst sollen die benötigten Datenstrukturen erstellt werden.

```
typedef vector <vector <vector <int> > > VVVI;
typedef vector <vector <int> > VVI;
typedef vector <int> VI;

ifstream fin ("ty.in");
ofstream fout ("ty.out");

VVVI nodeState; //erste Stufe: y, zweite Stufe: x,
                //dritte Stufe: Richtung,
                //Integer = Zustand (0 = unbesucht, 1 = besucht)
VVVI isSicher; //0 = unbesucht, 1 = unsicher, 2 = sicher;
VVI sicherN; //sichere Subknoten
VVI unsicherN; //unsichere Subknoten

VVVI unsicherM; //0 = nein, 1 = ja
VVI AdjM; //0 = frei, 1 = Mauer,
        //2 = Ausgang, 3 = unfrei, 4 = sicher
VVI dirs {{0, 1},{1, 0},{0, -1},{-1, 0}}; // y,x
        //Richtungen in die der Yamyam gehen kann

int width, height; // Höhe und Breite des Feldes
```

Das Programm liest die Eingabe von der Datei "ty.in", und schreibt die Ausgabe in "ty.out". Die Vektormatrix aus Vektoren *nodeState* erfüllt die Funktion von $\mathbf{B}^{\intercal}$. Da sich die Größe der Welt nicht ändert und wir nur "random access-Zugriffe ausführen müssen, verwenden wir den STL-Container vector, der diese

Operation in konstanter Zeit ausführen kann. Die Zeitkomplexität für seine Initialisierung ist linear zum alloziertem Speicher. Die Matrix *isSicher* erfüllt die Funktion von $\mathbf{S}^\intercal$. $\mathbf{AdjM}^\intercal$ wird definiert durch die Vektormatrix $AdjM$, die für *height*(Höhe der Welt) Reihen genau *width*(Breite der Welt) Vektoren enthält. In dieser Vektormatrix werden die angrenzenden Felder eines Feldes durch den Vektor *dirs* definiert, welcher für jede der 4 Richtungen die dahin benötigte Indexänderung relativ zum aktuellen Feld enthält.
sicherN und *unsicherN* enthalten die in Teil 1 und Teil 2 mittels rDFS ermittelten sicheren bzw. unsicheren Felder. Der Einfachheit halber wird angenommen, dass die äus quadratischen Feldern bestehende Weltäuch rechteckig ist. Einerseits wird dadurch die Eingabe einfacher, andererseits lässt sich immer jede erdenkliche nicht rechteckige Welt so konstruieren, indem man einfach Felder, die nicht existieren sollen, als Wand definiert.

```
int main()
{
    //Eingabe
    fin >> width >> height;
    AdjM.resize (height, VI (width));
    unsicherM.resize (height, VVI (width, VI (4, 0)));
    nodeState.resize (height, VVI (width, VI (4, 0)));
    isSicher.resize (height, VVI (width, VI (4, 0)));
    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            fin >> AdjM[i][j];
            if (AdjM[i][j] == 2) { //sichere Felder markieren
                sicherN.push_back (vector <int> {i, j});
                for (int d = 0; d < 4; ++d)
                    isSicher[i][j][d] = 2;
            }
        }
    }
```

Die Eingabe erfolgt über die erwähnten Dateien. Hier werden auch die Matrizen initialisiert. Die unterschiedlichen Werte, die sie annehmen können und ihre Bedeutung, lassen sich aus aus den Kommentaren in Abbildung 6 entnehmen. Im Folgenden sei ein Vergleich des BWINF-Eingabeformats und meines Eingabeformats dargestellt.

Im BWINF-Format stellt jedes Zeichen ein Feld dar, in meinem Format jede Nummer. Eine 1 ist gleich einer #, eine 0 gleich einem ' ' und eine 2 gleich einem E. Zudem ist in meinem Format bereits die Dimension des Rechtecks vorangestellt. Dieses Format hatte ich bereits vor der Veröffentlichung der Beispiele entwickelt und wollte es danach nicht mehr aufgeben (wodurch besagter Konvertierer entstand). Im Code dienen die beiden äußeren for-Schleifen der Initialierung von $AdjM$, welche einfach die in der Eingabe vorkommenden Werte enthält. Wird ein Ausgang eingelesen (if-Klammer), so ist dieses Feld von allen Richtungen als sicher zu betrachten (innerster for-Loop). Durch diese isSicher-

Beispielwelt 1 - unterschiedliche Formate

```
12 8
1 1 1 1 1 1 1 1 1 1 1 1
1 0 1 0 0 0 0 0 1 0 1 1
1 0 0 0 1 1 0 0 0 0 0 1
1 0 0 0 1 0 0 0 2 1 0 1
1 0 0 0 0 0 0 0 1 1 0 1
1 0 0 0 0 0 0 0 0 0 0 1
1 0 1 2 0 0 0 0 2 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
```

```
############
# #      # #
#   ##     #
#   #   E# #
#       ## #
#          #
# #E    E###
############
```

Abbildung 7: Links: Mein Format, Rechts: BWINF-Format

Initialisierung erhalten die Tiefensuchen eine Wertebasis, auf der sie aufbauen können.

Nach der Einlesung der Daten beginnt der Algorithmus mit dem Bestimmen aller unsicheren Felder.

```
//Felder markieren, von denen man nie ein sicheres Feld erreichen wird.
for (int i = 0; i < sicherN.size(); ++i) {
    for (int d = 0; d < 4; ++d) {
        fout << endl;
        //alle nicht unsicheren Subknoten markieren
        dfs (sicherN[i][0], sicherN[i][1], d, 0);
    }
}
```

Die äußerste for-Schleife iteriert dabei über alle aus der Eingabe bekannten Ausgangsknoten, um von ihnen rDFS durchzuführen. Die innere for-Schleife führt rDFS dann von jedem Subknoten des Ausgangsknoten durch, also jede Richtung. Die "2ïst die "Magic Number": Trifft die Tiefensuche auf sie (also auf einen Ausgang), ist das Feld nicht unsicher. Anschließend wird die rDFS mit der Funktion *dfs* gestartet. Die ersten zwei Parameter sind dabei die y- und x-Koordinaten des Subknotens, der dritte die Richtung, in die der Knoten gehen würde. Für diese gilt es ja dann, einen passenden vorherigen Subknoten zu finden. Die 0 gibt an, das eine rDFS von Ausgangsfeldern aus durchgeführt wird. Dadurch kann *dfs* die Status der Knoten in *isSicher* entsprechend setzen.

```
for (int i = 0; i < height; ++i) {
    for (int j = 0; j < width; ++j) {
        for (int d = 0; d < 4; ++d) {
            if (nodeState[i][j][d] == 0 && AdjM[i][j] == 0) {
                //aus Ergebniss der rDFS alle unsicheren Subknoten ermitteln
                unsicherN.push_back (vector <int> {i, j, d});
                unsicherM[i][j][d] = 1;
                isSicher[i][j][d] = 1;
            }
        }
    }
}
```

Der vorherige Code hat in *nodeState* alle besuchten Knoten markiert. Diese Ergebnisse werden jetzt ausgewertet, um alle unsicheren Subknoten zu finden.

Über diese iterieren die drei for-Schleifen. Die if-Abfrage prüft dann, ob der entsprechende Subknoten unsicher ist (wenn nodeState = 0 ist) und keine Wand oder Ausgang ist (wenn Adjm = 0 ist). Ist das der Fall, wird der Subknoten der Liste *unsicherN* der unsicheren Knoten hinzugefügt. Die Zustandsmatrizen *unsicherM* und *isSicher* markieren den Subknoten als unsicher (1 = unsicher).

```
nodeState.clear();
nodeState.resize (height, VVI (width, VI (4, 0)));
//Felder markieren, von denen man 100%-ig ein sicheres Feld erreiche
for (int i = 0; i < unsicherN.size(); ++i) {
    for (int d = 0; d < 4; ++d) {
        dfs (unsicherN[i][0], unsicherN[i][1], unsicherN[i][2], 1);
    }
}
```

Als nächstes wird Teil 2 des Algorithmus durchgeführt. Dieser nimmt die Subknoten die Liste *unsicherN* als Startsubknoten, um alle Subknoten herauszufinden, die einen unsicheren Subknoten erreichen können. Dazu resetten wir zunächst den Besucht-Vektor *nodeState*, alle Subknoten könne wieder besucht werden. Der Rest funktioniert wie die Umsetzung des Teil 1, nur eben mit dem Unterschied der Startknoten. Die 1 als letzter Parameter von *dfs* signalisiert der rDFS, das jetzt Teil 2 stattfindet.

```
int dfs (int y, int x, int d, int state) {
    fout << "visiting " << x << ' ' << y << ' ' << d << endl;
    nodeState[y][x][d] = 1; //siehe Status nodeState
    if (state == 1) {
        isSicher[y][x][d] = 1;
    }
    int d1 = (d-1) < 0 ? 3 : (d-1); //Die RRichtungen, in die man geh
    int d2 = (d+1) > 3 ? 0 : (d+1);
    int d3 = (d+2) > 3 ? (d-2) : (d+2);
    int nextX = x-dirs[d][1]; //Eins der Felder, von denen man potent
    int nextY = y-dirs[d][0];
    int nextX1 = x-dirs[d1][1];  //auch die Richtung im Array davor u
    int nextY1 = y-dirs[d1][0];
    int nextX2 = x-dirs[d2][1]; //und von danach
    int nextY2 = y-dirs[d2][0];
    if (AdjM[nextY][nextX] == 0 && nodeState[nextY][nextX][d] == 0) {
        dfs (nextY, nextX, d, state);
    }
    if (AdjM[nextY][nextX] == 1 && nodeState[y][x][d3] == 0) { //oder
        dfs (y, x, d3, state);
    }
    if (AdjM[nextY1][nextX1] == 0 && AdjM[nextY2][nextX2] == 1
        && nodeState[nextY1][nextX1][d1] == 0) { //wenn das Yamyam vo
        dfs (nextY1, nextX1, d1, state);
    }
    if (AdjM[nextY2][nextX2] == 0 && AdjM[nextY1][nextX1] == 1
        && nodeState[nextY2][nextX2][d2] == 0) { //wenn das Yamyam vo
        dfs (nextY2, nextX2, d2, state);
    }
    return 0;
}
```

Beide Teile verwenden die Funktion dfs, die rDFS implementiert. als Parameter bekommt sie den Knoten yxd übergeben (y=Reihe,x=Spalte,d=Richtung oder Subknoten). *state* gibt die Art der rDFS an. Ist er 1, wird also Teil 2 durchgeführt, markiert die erste if-Abfrage den Knoten in *isSicher* als nicht sicher. Nur in Teil 2 können wir nämlich definitiv schon in der rDFS sagen, dass ein Knoten nicht sicher ist. Davor wird der jetzt besuchte Subknoten yxd in *nodeState* als solcher markiert. Der Rest der Funktion ruft rDFS für die Subknoten auf, von denen aus man yxd erreichen kann. Die d- und next-Variablen geben für das angrenzende Feld (x_n, y_n) die Richtung d_n an, mit der man von diesem Feld kommen müsste, um auf yxd zu kommen. Die entsprechenden Paare bilden dann (nextY, nextX, d), (y, x, d3), (nextY1, nextX1, d1) und (nextY2, nextX2, d2). Die vier if-Abfragen realisieren den in Abbildung 4 dargestellten Test, ob man von einem Feld kommen kann. Dabei setzt die erste if-Abfrage Fall 1 um, die zweite Fall 2 usw. Nehmen wir uns z.B. die erste if-Abfrage, haben wir den Fall, dass der zu untersuchende Knoten $V(n)$ genau zur Richtung d von yxd passt, er liegt nämlich hinter diesem Feld. Deswegen ist die Bedingung für die if-Abfrage, dass $V(n)$ frei ist . $AdjM$ muss an dieser Stelle deswegen 0, also keine Wand oder Ausgang sein. Die Abfrage von *nodeState* testet, ob das Feld schon besucht ist. Falls nein, kann rDFS für (nextY, nextX, d) ausgeführt werden. Die anderen if-Abfragen funktionieren äquivalent. Als Standardwert liefert die Funktion 0 zurück.

Nachdem wir beide Teile des Algorithmus ausgeführt haben, haben wir genügend Daten gesammelt, um die Lösung auszugeben:

```
//0 = Wand, 1 = unsicher, 2 = fraglich, 3 = sicher, 4 = ausgang, 5 = würde
for (int i = 0; i < height; ++i) {
    for (int j = 0; j < width; ++j) {
        if (AdjM[i][j] == 1) { //Wenn Feld Wand, dann so ausgeben
            fout << "0 0 0 0 ";
        } else if (AdjM[i][j] == 2) { //Wenn Feld Ausgang, dann so ausgeben
            fout << "4 4 4 4 ";
        } else {
            for (int d = 0; d < 4; ++d) {
                if (AdjM[i+dirs[d][0]][j+dirs[d][1]] == 1) { //Liegt das Fe
                    fout << "5 ";
                    continue;
                }
                if (isSicher[i][j][d] == 0) { //Wenn ein Subknoten nicht al
                    isSicher[i][j][d] = 2;
                }
                if (unsicherM[i][j][d] == 1) {
                    fout << "1 ";
                } else {
                    fout << (isSicher[i][j][d] == 2 ? '3' : '2') << ' ';
                }
            }
        }
        fout << ' ';
    }
    fout << endl;
}
```

Das Format der Ausgabe ist wie meines der Eingabe, mit dem Unterschied,

das statt einem Knoten alle seine vier Subknoten, durch ein Leerzeichen getrennt, ausgegeben werden. Die nächste Gruppe Subknoten wird dann durch zwei Leerzeichen von der vorherigen getrennt. Nach jeder Reihe endet eine Zeile der Ausgabe. Die beiden for-Schleifen iterieren durch alle Knoten. Ist der aktuelle in AdjM bereits als Ausgang oder Wand gekennzeichnet, wird er auch als solcher durch die ersten beiden if-Abfragen ausgegeben. Das Kommentar über der Ausgabe gibt die verschiedenen Arten von Subknoten an. Handelt es sich nicht um ein spezielles Feld, iterieren wir durch die verschiedenen Richtungen eines Knoten. Für jeden solchen Subknoten haben wird drei Möglichkeiten: Die erste if-Abfrage testet, ob das Yamyam in diese Richtung in eine Wand torkeln würde. Als Startposition ist das irrelevant, die sich dadurch ergebenden weiteren Torkel-Richtungen werden sowieso abgearbeitet. Darum wird der Subknoten in so einem Fall auch als 5 markiert. Wurde der Subknoten in $isSicher$ nicht als unsicher markiert, muss er sicher sein, da er sonst in Teil 1 oder Teil 2 als unsicher markiert worden wäre. Wurde der Knoten hingegen in $unsicherM$ in Teil 1 als unsicher markiert, soll das auch ausgegeben werden. Die letzte else-Klammer testet noch den Fall, das der Knoten zwar nicht in Teil 1 als unsicher, aber in Teil 2 als unsicher markiert wurde. Das bedeutet, das dieser Subknoten zwar einen Ausgang erreichen kann, aber nicht zwangsläufig muss, womit er fraglich ist.

1.2.2 Graphische Ausgabe ”Torkelnde Yamyams”

Das Tool zur Erzeugung einer graphischen Ausgabe der Lösungen von "Torkelnde Yamyamsïst programmtechnisch untinteressant, sein Code soll hier nicht dargestellt werden. Grob gesagt liest es die Ausgabedatei vom Programm ”Torkelnde Yamyams” ein, und erstellt eine SVG-Datei der Welt, in der die Felder entsprechend markiert sind. Seine Implementierung findet sich in der Datei yamyamgraphik.cpp.

1.2.3 Konvertierung der Formate ”Torkelnde Yamyams”

Das Tool zur Konvertierung des BWINF-Formats der Eingabe zu "Torkelnde Yamyamsïn mein Format ist programmtechnisch untinteressant, sein Code soll hier nicht dargestellt werden. Grob gesagt liest es die Eingabedatei mit dem BWINF-Format ein und erzeugt eine Eingabedatei mit meinem Format. Seine Implementierung findet sich in der Datei yamyamformatkonvertierer.cpp.

1.2.4 Programm für die Erweiterung zur Berechnung des generalisierten Yamyam-Graphs

Als nächstes sei noch kurz die Implementierung für die verallgemeinerte Version des Problems vorgestellt. Diese ähnelt stark der des normalen Programms, lediglich die Datenstrukturen unterscheiden sich. Darauf soll hier auch der Fokus gelegt werden.

```
vector <map <int, int> > nodeState, nodeStateInit, /
            isSicher, isUnsicher; // 0 = unbesucht,
vector <map <int, VI> > AdjL, AdjLG; //1. Knoten, 2.
vector <bool> isExit; //Ist Knoten x ein Ausgang?
VI sicherN;
VVI unsicherN;
int nSize; //Anzahl der Knoten
```

Die Speicherstrukturen sind im Prinzip dieselben, man kann jedoch erkennnen, das die Matrizen mit der erwähnten Map der STL umgesetzt werden. Auch äquivalent zum ersten Programm erfolgt die Eingabe, nur das hier ein anderes Format zum Einsatz kommt. Das besteht wieder nur aus Zahlen und sieht folgendermaßen aus:

1. Zeile 1: N_1 - Anzahl der Knoten
   Anschließend folgen N_1 Knotenbeschreibungen, von denen jede so aufgebaut ist:
   (a) Zahl 1: Ist Knoten ein Ausgang? 0 - nein, 1 - nein
   (b) Zahl 2: N_2 - Zu wie vielen Knoten kann man von diesem Knoten gehen?
   Anschließend folgen N_2 Kantenbeschreibungen, von denen jede so aufgebaut ist:
      i. Zahl 1: Zu welchem Knoten kann ich gehen?

ii. Zahl 2: N_3 - Von wie vielen Knoten kann ich kommen, um meine Torkelei zu die Knoten fortsetzen zu können?
Anschließend folgen N_3 Knoten.

Dieses Format, wie auch sein Einlesen ist beides recht kompliziert, bildet jedoch keinen algorithmischen Mehrwert, weswegen ich den Code dem Leser ersparen möchte :)

```
for (int i = 0; i < sicherN.size(); ++i) {
    int uN = sicherN[i];
    for (auto d = AdjL[uN].begin(); d != AdjL[uN].end(); ++d) {
        for (int j = 0; j < (*d).second.size(); ++j) {
            dfs (uN, (*d).second[j], 0);
        }
    }
}
```

Diese for-Schleife implementiert Teil 1 des Algorithmus. Vom Aufbau ist sie fast identisch wie die des ersten Programms, auch hier werden alle sicheren Knoten im Vektor *sicherN* durchiteriert. Man beachte jedoch, dass das zweite Argument von *dfs* aus der Liste *VDavor* genommen wurde - hier kommt die neue Speicherstruktur zum Einsatz. Wie im ersten Programm handelt es sich jedoch letzten Endes um einen Knoten, der eine Verbindung zum ersten Argument *uN* hat. Teil 2 wird fast identisch zum ersten Programm umgesetzt, nicht jedoch die Funktion *dfs*.

```
int dfs (int node, int d, int state) { // Herausfinden, von wel
    nodeState[node][d] = 1; //siehe Status nodeState
    if (state == 1) {
        isSicher[node][d] = 1;
    }
    for (int i = 0; i < AdjL[d][node].size(); ++i) {
        int formerN = AdjL[d][node][i];
        if (nodeState[d][formerN] == 0 && isExit[d] == false) {
            dfs (d, formerN, state);
        }
    }
    return 0;
}
```

Wieder werden zu Beginn der Funktion die Zustände gesetzt. Es gilt herauszufinden, von welchen Knoten aus ich auf Knoten *node* über Knoten *d* gelangen kann. Im Gegensatz zum vorherigen Algorithmus geht das schnell, da *AdjL* (äquivalent zu *AdjM*) einen Vektor mit all diesen Knoten liefern kann: Über diesen wird iteriert und anschließend dfs für all diese Knoten aufgerufen, sollten sie noch nicht besucht sein.

```
for (int i = 0; i < nSize; ++i) {
    if (isSicher[i][i] != 1){
        fout << i << endl;
    }
}
```

Diese for-Schleife ist die Ausgabefunktion, die nach einem eleganten Prinzip arbeitet: *isSicher* speichert ja für jeden eingehenden Knoten eines jeden Knotens, ob diese Verbindung sicher oder unsicher ist. Das heißt, ob jede nach dieser Verbindung erreichbare Kante zu einem anderen Knoten sicher oder unsicher ist. Nun ist es eine logische Vorgabe für jede Eingabe des generalisierten Algorithmus, dass jeder Knoten eine Kante zu sich selbst aufweisen muss, nach deren Begehung er alle anderen Kanten besuchen kann. Diese Kante zu sich selbst lässt sich von keinem anderen Knoten aus erreichen und wird benötigt, wenn eine rDFS von diesem Knoten startet: Schließlich muss man davon ausgehen, dass das Yamyam anfangs in alle Richtungen torkeln kann. Um nun zu überprüfen, ob ein Knoten sicher ist, reicht es, sich diese eine Kante zu sich selbst anzuschauen ($isSicher[i][i]$), da in ihr ja Verbindungen zu allen anderen von diesem Knoten aus erreichbaren Knoten enthalten sind. Die Liste der sicheren Knoten wird in eine Datei ausgegeben. Zudem folgt dieser Ausgabe noch Code, der die Ergebnisse graphisch aufbereitet, in Form eines Dokuments in der dot-Sprache. Die entstandene .gv-Datei lässt sich mit dem Tool dot in eine pdf-Datei umwandeln, in der man dann den Graphen graphisch betrachten kann, wie z.B. in Abbildung 5 sehen kann.

1.2.5 Programm zur Beispielgenerierung für den generalisierten Algorithmus

Das Tool "Yamyam Konvertierer"konvertiert zur Prüfbarkeit des generalisierten Algorithmus aus einer Eingabe des ersten Programms eine Eingabe für diesen. Sowohl die Eingabe als auch die Ausgabe des Programms erfolgt in einer Datei.

1.3 Beispiele

Für alle Beispiele gebe ich aufgrund der Verständlichkeit nicht die numerischen Dateien, sondern die graphischen Ausgaben meiner Programme an. Diese Beispiele mit ihren numerischen Ergebnissen sowie weitere Beispiele, die zu groß sind, um hier abgebildet zu werden, finden sich im Ordner Beispiele. Alle Zeit- und Speicherangaben sind lediglich Orientierungswerte, die erst bei großen Feldern eine Rolle spielen.

1.3.1 Erstes Programm

Für die Eingabe der Beispiele wird hier nicht das BWINF- sondern mein Format dargestellt. Die Resultate sind jedoch identisch, schließlich wurde die Eingabedatei ja konvertiert. Die Ausgabe richtet sich nach dem Format beschrieben in Abbildung 1. Die sicheren Felder sind grün umrandet.
Die Eingabedateien benennen sich nach folgendem Muster: beispiel[1-9].in
Die Ausgabedateien benennen sich nach folgendem Muster:
beispiel[1-9].out
beispiel[1-9].svg

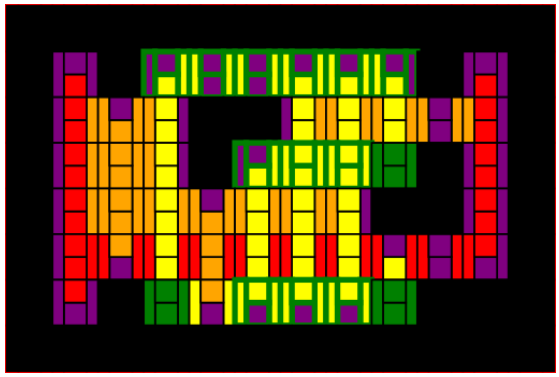
Beispiel 1

Beschreibung: BWINF-yamyams0.txt Beispiel

Eingabe:

```
############
#  #      #  #
#     ##     #
#     #    E#  #
#          ##  #
#              #
#  #E     E###
############
```

Ausgabe:



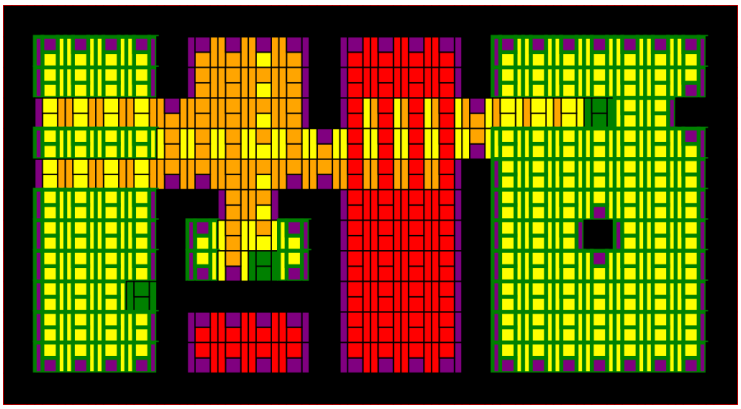
Zeit & Speicher: 0.012s, 0.4Mbyte

Beispiel 2

Beschreibung: BWINF-yamyams1.txt Beispiel

Eingabe:

```
24 13
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 1
1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 2 0 0 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1
1 0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 1
1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 1
1 0 0 0 0 1 0 0 2 0 1 0 0 0 0 1 0 0 0 0 0 0 0 1
1 0 0 0 2 1 1 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 1
1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 1
1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Ausgabe: 

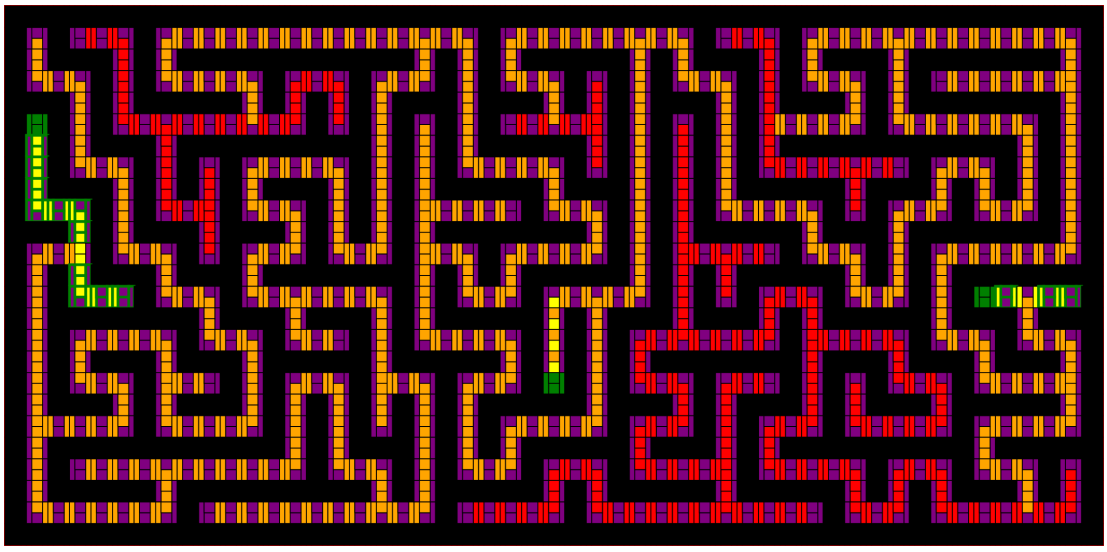
Zeit & Speicher: 0.016s, 0.4Mbyte

Beispiel 3

Beschreibung: BWINF-yamyams2.txt Beispiel

Eingabe:

```
51 25
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1 1 1 1 1 1 0 1
1 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 0 0 1
1 1 1 0 1 0 1 1 1 1 1 0 1 0 1 0 1 0 1 1 1 0 1 1 1 0 1 0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1 1 1 1 1
1 2 1 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 0 1
1 0 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1 1 0 1 0 1 0 1 0 1 0 1 1 1 1 1 1 1 1 1 1 0 1
1 0 1 0 0 0 1 0 1 0 1 0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 1 0 0 0
1 0 1 1 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 1 1 1 1 1 0 1 1 1 0 1 0 1 0 1 0 1 1 1 1 1 1 0 1 1 1 0 1
1 0 0 0 1 0 1 0 1 0 1 0 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
1 1 1 0 1 0 1 1 1 0 1 1 1 0 1 0 1 0 1 0 1 1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1 1 0 1 1 1 0 1 1 1 1 1 1
1 0 0 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 1
1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1
1 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 1 0 1 0 1 0 0 0 1 0 0 0 1 0 1 2 0 0 1
1 0 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 0 1 0 1 1 1 1 1 0 1 0 1 1 1 0 1 1 1 0 1 0 1 1 1 1 1 0 1 1 1 0 1
1 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 1
1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1 0 1 0 1 0 1 1 1 1 0 1 0 1 1 1 1 1 1 1 0 1
1 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 1 0 1 0 1 0 1 0 2 1 0 1 0 0 0 1 0 0 0 1 0 1 0 1 0 0 0 1 0 0 0 0 1
1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 1 1 1 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1 1 1 0 1 0 1
1 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 1
1 0 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 0 1 0 1 0 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1
1 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 1 1
1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1 0 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 0 1 0 1 1 1
1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Ausgabe: 

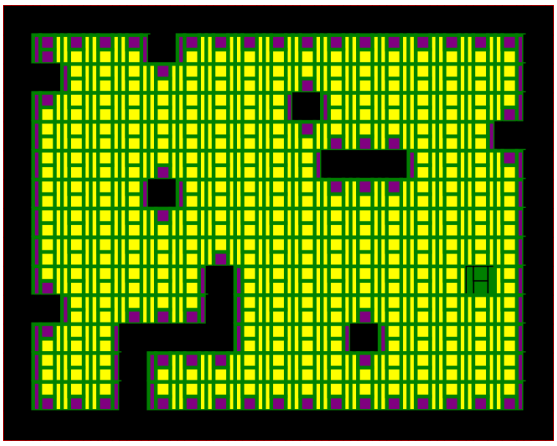
Zeit & Speicher: 0.038s, 0.4Mbyte

Beispiel 4

Beschreibung: BWINF-yamyams3.txt Beispiel

Eingabe:

```
19 15
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1
1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
1 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 1
1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 2 0 1
1 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 1 1 1 1 0 0 0 0 1 0 0 0 0 0 1
1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Ausgabe: 

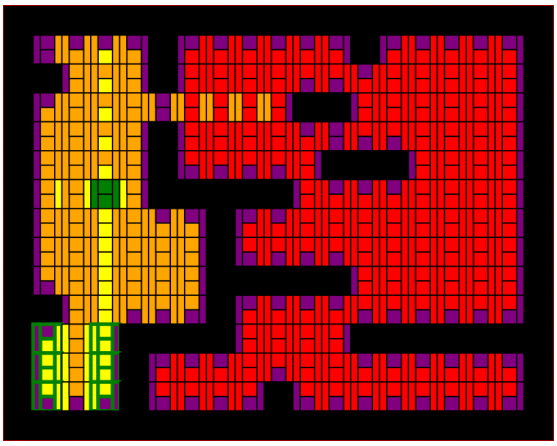
Zeit & Speicher: 0.028s, 0.4Mbyte

Beispiel 5

Beschreibung: BWINF-yamyams4.txt Beispiel

Eingabe:

```
19 15
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 1
1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1
1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 1 0 0 0 0 0 1 1 1 0 0 0 0 1
1 0 0 2 0 1 1 1 1 1 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 1
1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1
1 0 0 0 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1
1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Ausgabe: 

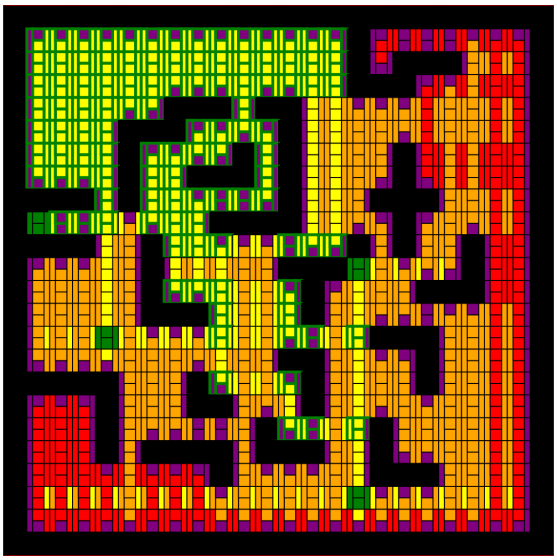
Zeit & Speicher: 0.031s, 0.4Mbyte

Beispiel 6

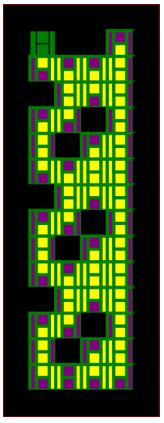
Beschreibung: BWINF-yamyams5.txt Beispiel

Eingabe:

```
24 24
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1
1 0 0 0 0 0 0 0 1 1 1 0 1 1 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 1
1 0 0 0 0 0 1 0 0 1 1 1 0 1 0 0 0 0 1 0 0 0 0 1
1 1 1 1 0 1 0 0 0 0 0 0 0 1 0 0 0 1 1 1 0 0 0 1
1 2 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 0 0 0 0 1
1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 1
1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 1 2 0 0 0 0 1 1
1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 1
1 0 0 0 0 0 0 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1
1 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 1
1 1 1 1 1 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1
1 0 0 0 1 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 1
1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1
1 0 0 0 1 0 1 1 1 1 0 1 1 1 0 0 1 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 1 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Ausgabe: 
Zeit & Speicher: 0.012s, 0.4Mbyte

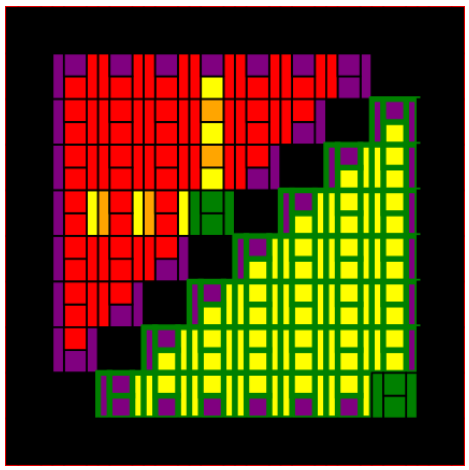
Beispiel 7
Beschreibung: BWINF-yamyams6.txt Beispiel

Eingabe:
```
6 16
1 1 1 1 1 1
1 2 1 1 0 1
1 0 0 0 0 1
1 1 0 0 0 1
1 0 0 1 0 1
1 0 1 0 0 1
1 0 0 0 0 1
1 1 0 0 0 1
1 0 0 1 0 1
1 0 1 0 0 1
1 0 0 0 0 1
1 1 0 0 0 1
1 0 0 1 0 1
1 0 1 0 0 1
1 0 0 0 0 1
1 1 1 1 1 1
```
Ausgabe: 
Zeit & Speicher: 0.016s, 0.4Mbyte

Beispiel 8
Beschreibung: Erstes eigenes Beispiel, 56 begehbare Felder, Weltgröße: 10x10

Eingabe:
```
10 10
1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0 1 1
1 0 0 0 0 0 0 1 0 1
1 0 0 0 0 0 1 0 0 1
1 0 0 0 2 1 0 0 0 1
1 0 0 0 1 0 0 0 0 1
1 0 0 1 0 0 0 0 0 1
1 0 1 0 0 0 0 0 0 1
1 1 0 0 0 0 0 0 2 1
1 1 1 1 1 1 1 1 1 1
```

Ausgabe:



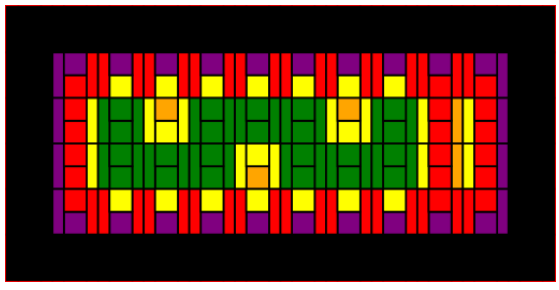
Zeit & Speicher: 0.016s, 0.4Mbyte

Beispiel 9

Beschreibung: Zweites eigenes Beispiel, 40 begehbare Felder, Weltgröße: 12x6

Eingabe:
```
12 6
1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0 0 0 0 1
1 0 2 0 2 2 2 0 2 0 0 1
1 0 2 2 2 0 2 2 2 0 0 1
1 0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1
```

Ausgabe:



Zeit & Speicher: 0.012s, 0.4Mbyte

Beispiel 10
Beschreibung: Drittes eigenes Beispiel, 801.790 begehbare Felder, Weltgröße: 1000x1000
Eingabe- und Ausgabedatei sind zu groß, um in der Dokumentation abgebildet werden zu können und liegen anbei. Eingabe: ex3.in, Ausgabe: ex3??>?/».?>?out
Zeit & Speicher: 2.048s, 150Mbyte

1.3.2 Generalisiertes Programm

Als Beispiele werden hier die "vergraphtenËingaben der BWINF-Beispiele 1 und 2, das Beispiel 10 des konkreten Programms, sowie zwei weitere Graphen genommen. Die Ein- und Ausgabedateien sind sehr lang, weswegen ich hier für die Rechteck-Welten wieder nur das normale Format angebe.
Die Eingabedateien benennen sich nach folgendem Muster: beispielA[1-5].in
Die Ausgabedateien benennen sich nach folgendem Muster:
beispielA[1-5].out
beispielA[1-5].gv
beispielA[1-5].png

Beispiel 1
Beschreibung: BWINF-yamyams0.txt, konvertiert in generalisiertes Format

```
###########
# #     # #
#   ##    #
#   #   E# #
#       ## #
#          #
# #E    E###
###########
```

Eingabe:
Zeit & Speicher: 0.012s, 0.4Mbyte

Beispiel 2
Beschreibung: BWINF-yamyams1.txt, konvertiert in generalisiertes Format

Eingabe:

```
24 13
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1
1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 2 0 0 1 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1
1 0 0 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1
1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 0 0 1
1 0 0 0 0 1 0 0 2 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1
1 0 0 0 2 1 1 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1
1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1
1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Zeit & Speicher: 0.012s, 0.4Mbyte

Beispiel 3

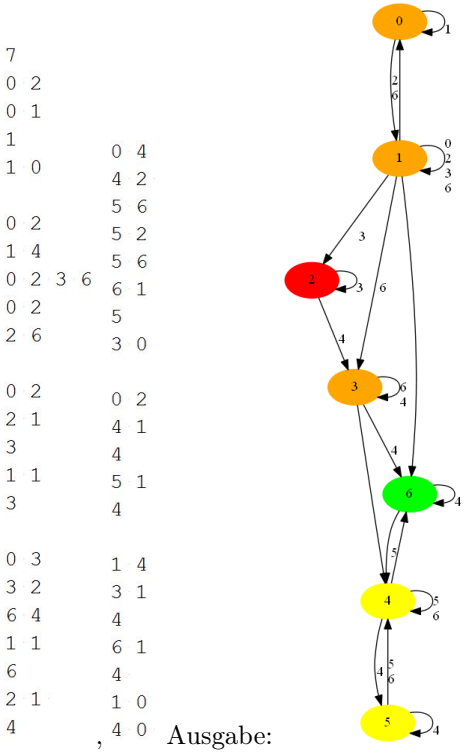
Beschreibung: BWINF-yamyams6.txt, konvertiert in generalisiertes Format

Eingabe:

```
24 24
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 1
1 0 0 0 0 0 0 1 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 1
1 0 0 0 0 1 0 0 1 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0 1
1 1 1 1 0 1 0 0 0 0 0 0 1 0 0 0 1 1 1 0 0 0 0 0 1
1 2 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0 1
1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 1 1
1 0 0 0 0 0 1 0 0 0 0 0 1 1 1 2 0 0 0 0 1 0 0 1 1
1 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 1 1 0 0 1 1
1 0 0 0 0 0 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1
1 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 1
1 1 1 1 1 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1
1 0 0 0 1 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 1
1 0 0 0 1 0 1 1 1 1 0 1 1 1 0 0 1 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

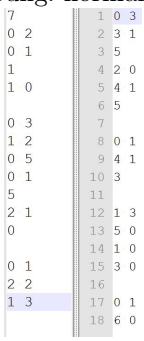
Zeit & Speicher: 0.012s, 0.4Mbyte

Beispiel 4
Beschreibung: normaler Graph, viele Kanten

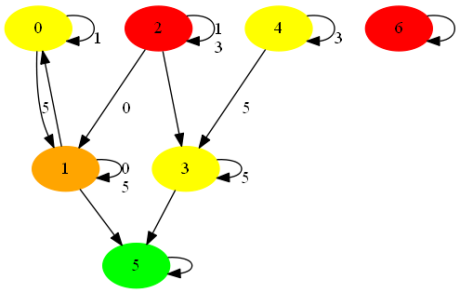


Zeit & Speicher: 0.100s, 0.5Mbyte

Beispiel 5
Beschreibung: normaler Graph, wenige Kanten



Eingabe:



Ausgabe:
Zeit & Speicher: 0.012s, 0.4Mbyte

2 Anhang

2.1 Quelltext

Torkelnde Yamyams

```
#include <iostream>
#include <fstream>
#include <vector>

using namespace std;

typedef vector <vector <vector <int> > > VVVI;
typedef vector <vector <int> > VVI;
typedef vector <int> VI;

ifstream fin ("ty.in");
ofstream fout ("ty.out");

VVVI nodeState; //erste Stufe: y, zweite Stufe: x,
              //dritte Stufe: Richtung,
              //Integer = Zustand (0 = unbesucht, 1 = besucht)
VVVI isSicher; //0 = unbesucht, 1 = unsicher, 2 = sicher;
VVI sicherN; //sichere Subknoten
VVI unsicherN; //unsichere Subknoten

VVVI unsicherM; //0 = nein, 1 = ja
VVI AdjM; //0 = frei, 1 = Mauer,
       //2 = Ausgang, 3 = unfrei, 4 = sicher
VVI dirs {{0, 1},{1, 0},{0, -1},{-1, 0}}; // y,x
       //Richtungen in die der Yamyam gehen kann

int width, height; // HÃ¶he und Breite des Feldes

//dsf: findet alle Subknoten, die Subknoten yxd besuchen kÃ¶nnen
int dfs (int y, int x, int d, int state, int depth) {
```

```
  //  cout << depth << endl;
    nodeState[y][x][d] = 1; //siehe Status nodeState
    if (state == 1) {
        isSicher[y][x][d] = 1;
    }
    int d1 = (d-1) < 0 ? 3 : (d-1); //Die Richtungen, in die man gehen
        mÃ¼sste, um potentiell yxd zu erreichen
    int d2 = (d+1) > 3 ? 0 : (d+1);
    int d3 = (d+2) > 3 ? (d-2) : (d+2);
    int nextX = x-dirs[d][1]; //Eins der Felder, von denen man
        potentiell yxd erreichen kÃ¶nnte
    int nextY = y-dirs[d][0];
    int nextX1 = x-dirs[d1][1]; //auch die Richtung im Array davor
        untersuchen
    int nextY1 = y-dirs[d1][0];
    int nextX2 = x-dirs[d2][1]; //und von danach
    int nextY2 = y-dirs[d2][0];
    if (AdjM[nextY][nextX] == 0 && nodeState[nextY][nextX][d] == 0) {
        //wenn das Yamyam von dort kommen kann, untersuche das Feld
        dfs (nextY, nextX, d, state, depth+1);
    }
    if (AdjM[nextY][nextX] == 1 && nodeState[y][x][d3] == 0) { //oder
        von dort
        dfs (y, x, d3, state, depth+1);
    }
    if (AdjM[nextY1][nextX1] == 0 && AdjM[nextY2][nextX2] == 1
        && nodeState[nextY1][nextX1][d1] == 0) { //wenn das Yamyam von
            dort kommen kann, untersuche das Feld
        dfs (nextY1, nextX1, d1, state, depth+1);
    }
    if (AdjM[nextY2][nextX2] == 0 && AdjM[nextY1][nextX1] == 1
        && nodeState[nextY2][nextX2][d2] == 0) { //wenn das Yamyam von
            dort kommen kann, untersuche das Feld
        dfs (nextY2, nextX2, d2, state, depth+1);
    }
    return 0;
}

int main()
{
    //Eingabe
    fin >> width >> height;
    AdjM.resize (height, VI (width));
    unsicherM.resize (height, VVI (width, VI (4, 0)));
    nodeState.resize (height, VVI (width, VI (4, 0)));
    isSicher.resize (height, VVI (width, VI (4, 0)));
    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            fin >> AdjM[i][j];
            if (AdjM[i][j] == 2) { //sichere Felder markieren
```

```
            sicherN.push_back (vector <int> {i, j});
            for (int d = 0; d < 4; ++d)
                isSicher[i][j][d] = 2;
        }
    }
}
cout << "initialisiert!" << endl;
//Felder markieren, von denen man nie ein sicheres Feld erreichen
    wird.
for (int i = 0; i < sicherN.size(); ++i) {
    for (int d = 0; d < 4; ++d) {
        //alle nicht unsicheren Subknoten markieren
        dfs (sicherN[i][0], sicherN[i][1], d, 0, 0);
    }
}
for (int i = 0; i < height; ++i) {
    for (int j = 0; j < width; ++j) {
        for (int d = 0; d < 4; ++d) {
            if (nodeState[i][j][d] == 0 && AdjM[i][j] == 0 &&
                AdjM[i+dirs[d][0]][j+dirs[d][1]] != 1) {
                //aus Ergebniss der rDFS alle unsicheren Subknoten
                    ermitteln
                unsicherN.push_back (vector <int> {i, j, d});
                unsicherM[i][j][d] = 1;
                isSicher[i][j][d] = 1;
            }
        }
    }
}
nodeState.clear();
nodeState.resize (height, VVI (width, VI (4, 0)));
//Felder markieren, von denen man 100%-ig ein sicheres Feld
    erreichen wird.
for (int i = 0; i < unsicherN.size(); ++i) {
    for (int d = 0; d < 4; ++d) {
        dfs (unsicherN[i][0], unsicherN[i][1], unsicherN[i][2], 1,
            0); //markiere alle als sicher
    }
}
//Ausgabe
//0 = Wand, 1 = unsicher, 2 = fraglich, 3 = sicher, 4 = ausgang, 5 =
    wÃ¼rde zu Wand torkeln, nicht mÃ¶glich
for (int i = 0; i < height; ++i) {
    for (int j = 0; j < width; ++j) {
        if (AdjM[i][j] == 1) { //Wenn Feld Wand, dann so ausgeben
            fout << "0 0 0 0 ";
        } else if (AdjM[i][j] == 2) { //Wenn Feld Ausgang, dann so
            ausgeben
            fout << "4 4 4 4 ";
        } else {
```

```
                for (int d = 0; d < 4; ++d) {
                    if (AdjM[i+dirs[d][0]][j+dirs[d][1]] == 1) { //Liegt
                        das Feld an einer Wand, soll der dahin verbundene
                        Subknoten als nicht eindeutig markiert werden
                        fout << "5 ";
                        continue;
                    }
                    if (isSicher[i][j][d] == 0) { //Wenn ein Subknoten
                        nicht als unsicher markiert wurde, soll er als
                        sicher gehandhabt werden
                        isSicher[i][j][d] = 2;
                    }
                    if (unsicherM[i][j][d] == 1) {
                        fout << "1 ";
                    } else {
                        fout << (isSicher[i][j][d] == 2 ? '3' : '2') << '
                            ';
                    }
                }
            }
            fout << ' ';
        }
        fout << endl;
    }
    //char c;
    //cin >> c;
}
```

Torkelnde Yamyams Abstrakt

```
#include <iostream>
#include <fstream>
#include <vector>
#include <map>

using namespace std;

typedef vector <vector <vector <int> > > VVVI;
typedef vector <vector <int> > VVI;
typedef vector <int> VI;

#define fakeN 1000000000

ifstream fin ("ty.in");
ofstream fout ("ty.out");
ofstream graph ("graph.gv");

vector <map <int, int> > nodeState, nodeStateInit, //erste Stufe:
    Knoten: Richtung, Integer = Zustand (0 = unbesucht, 1 = besucht)
```

```
        isSicher, isUnsicher; // 0 = unbesucht, 1 = unsicher, 2 = sicher
vector <map <int, VI> > AdjL, AdjLG; //1. Knoten, 2. verbundene Knoten
    (Richtung, wohin will ich danach), 3. wenn ich zu diesem Knoten
    gehen will, von welchen kann ich kommen?
vector <bool> isExit; //Ist Knoten x ein Ausgang?
VI sicherN;
VVI unsicherN;
int nSize; //Anzahl der Knoten

//dsf-status: 0 - nicht gefunden, 1 - gefunden
int dfs (int node, int d, int state) { // Herausfinden, von welchen
    Feldern aus man Knoten node erreichen kann. d = von welchem Knoten
    komme ich
    nodeState[node][d] = 1; //siehe Status nodeState
    if (state == 1) {
        isSicher[node][d] = 1;
    }
    for (int i = 0; i < AdjL[d][node].size(); ++i) {
        int formerN = AdjL[d][node][i];
        if (nodeState[d][formerN] == 0 && isExit[d] == false) {
            dfs (d, formerN, state);
        }
    }
    return 0;
}

int main()
{
    //Eingabe
    fin >> nSize;
    AdjL.resize (nSize);
    AdjLG.resize(nSize);
    nodeStateInit.resize (nSize);
    isExit.resize (nSize);
    for (int i = 0; i < nSize; ++i) {
        int cons, isE; //connections, isExit
        fin >> isE >> cons;
        isExit[i] = isE;
        if (isE == 1) {
            sicherN.push_back (i);
        }
        for (int j = 0; j < cons; ++j) {
            int rSize, tile; //number of reachable nodes when coming from
                node tile, state of
            fin >> tile >> rSize;
            vector <int> reachableV (rSize);
            for (int k = 0; k < rSize; ++k) {
                int reachable;
                fin >> reachable;
                AdjL[i][reachable].push_back (tile);
```

```
            reachableV[k] = reachable;
        }
        AdjLG[i].insert (make_pair(tile, reachableV));
        if (rSize == 0 && isE == 1) { //ist der Knoten ein Ausgang,
             sollen auch eingehende Kanten ohne weiterfÃ¼hrende
             Verbindung durchlaufen werden
            AdjL[i][fakeN].push_back (tile);
        }
        nodeStateInit[i].insert (make_pair (tile, 0));
    }
}
isSicher = nodeStateInit;
nodeState = nodeStateInit;
for (int i = 0; i < isExit.size(); ++i) {
    if (isExit[i] == true) {
        for (auto it = isSicher[i].begin(); it != isSicher[i].end();
             ++it) {
            (*it).second = 2;
        }
    }
}
//FÃ¼r jeden Knoten und jede Richtung testen, ob sie sicher ist.
    for (int i = 0; i < sicherN.size(); ++i) {
        int uN = sicherN[i];
        for (auto d = AdjL[uN].begin(); d != AdjL[uN].end(); ++d) {
            for (int j = 0; j < (*d).second.size(); ++j) {
                dfs (uN, (*d).second[j], 0);
            }
        }
    }
for (int i = 0; i < nodeState.size(); ++i) {
    for (auto d = nodeState[i].begin(); d != nodeState[i].end();
         ++d) {
        int state = (*d).second;
        int dir = (*d).first;
        if (state == 0) {
            unsicherN.push_back (vector <int> {i, dir});
            isSicher[i][dir] = 1;
        }
    }
}
isUnsicher = isSicher;
nodeState = nodeStateInit;
for (int i = 0; i < unsicherN.size(); ++i) {
    dfs (unsicherN[i][0], unsicherN[i][1], 1);
}
//Ausgabe
for (int i = 0; i < nSize; ++i) {
    if (isSicher[i][i] != 1){
        fout << i << endl;
```

```
        }
    }
    graph << "digraph yamyam {" << endl;
    for (int i = 0; i < AdjLG.size(); ++i) {
        for (auto it = AdjLG[i].begin(); it != AdjLG[i].end(); ++it) {
                int con = (*it).first;
                graph << con << " -> " << i << " [label = \"";
                for (auto it2 = (*it).second.begin(); it2 !=
                     (*it).second.end(); ++it2) {
                    graph << "\\n" << *it2;
                }
                graph << "\"];" << endl;
        }
        bool unsicher = true;
        for (auto d = isUnsicher[i].begin(); d != isUnsicher[i].end();
             ++d) {
            unsicher = (*d).second == 1 ? unsicher : false;
        }
        graph << i << "[label = \"" << i << "\"";
        if (isExit[i] == true) {
            graph << " color=green style = filled";
        }
        else if (isSicher[i][i] != 1) {
            graph << " color=yellow style = filled";
        } else if (unsicher == true) {
            graph << " color=red style = filled";
        } else {
            graph << " color=orange style = filled";
        }
        graph << "];" << endl;
    }
    graph << "}" << endl;
}
```

2.2 Quellen

Zur Generierung der SVG-Datei habe ich die Bibliothek simple-svg von Mark Turney benutzt: `https://code.google.com/archive/p/simple-svg`