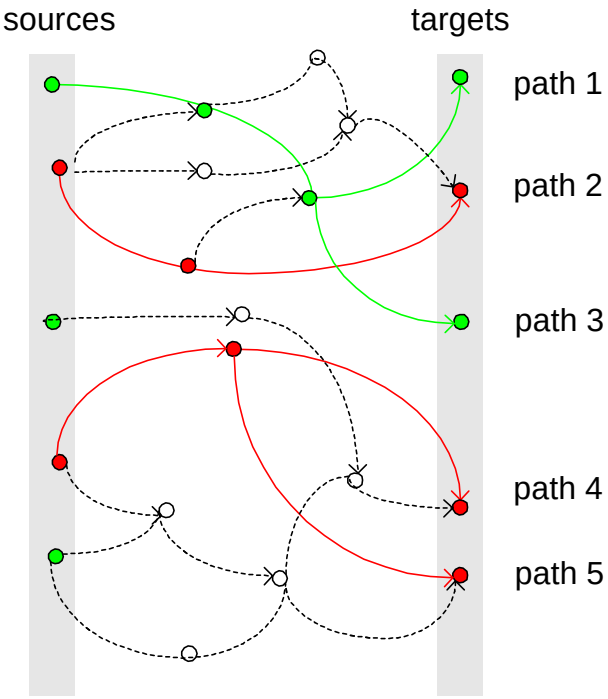


Das „Colored-Trees“-Problem

Version 1.1

Autor: Josef Hübl

Erstellt am: 08.09.2003

Geändert am: 01.06.2006

Von: Josef Hübl (Triple-S GmbH)

INHALTSVERZEICHNIS

Seite

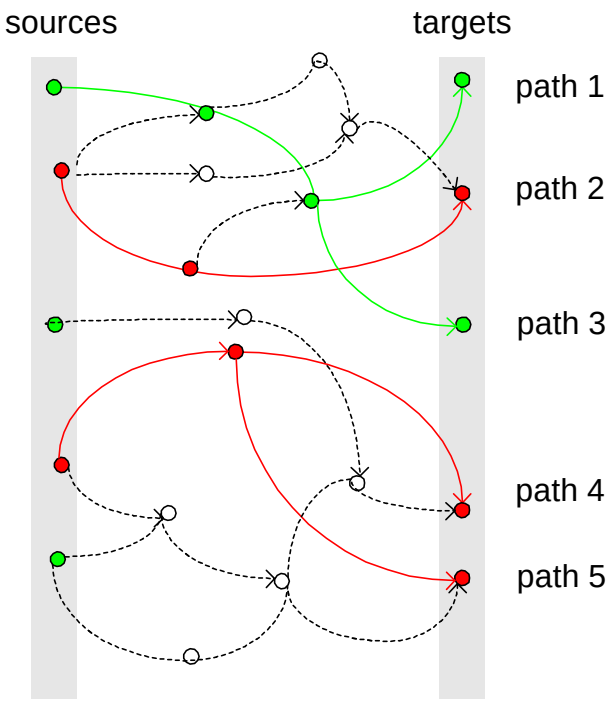
 Autor: Josef Hübl

1. Das „Colored-Trees“-Problem

1.1 Problembeschreibung

Gegeben ist ein gerichteter Graph (ohne Schleifen), sowie eine Menge von Eingangsknoten (sources) und eine Menge von Ausgangskonten (targets), wobei sowohl den Eingangs- als auch den Ausgangskonten je eine Farbe zugeordnet ist. Das „Colored-Tree“-Problem besteht darin, eine Menge von knoten-disjunkten Bäumen (trees) zu finden, so dass jeder Ausgangskonten Endknoten eines Baumes ist, der Start-Knoten jeden Baumes zu den Eingangsknoten und alle Endknoten zu den Ausgangsknoten gehören und dem Startknoten die selbe Farbe zugeordnet ist, wie allen Endknoten.

Da die gefunden Bäume knoten-disjunkt sind, können alle Knoten eines Lösungsbaumes gleich eingefärbt werden.



Gefärbte Bäume in einem gerichteten Graphen

Autor: Josef Hübl

1.2 Bezug zur Praxis

Sind die Knoten die Kontaktstellen und die Kanten die Verbindungen auf elektrischen Leiterplatten und werden den Eingängen bzw. Ausgängen bestimmte Signalarten zugeordnet, so liegt ein „Colored-Trees“-Problem vor, wenn ermittelt werden soll, welche Verbindungen unterbrochen werden müssen so, dass jeder Ausgang mit einem passenden signalgebenden Eingang verbunden ist und Kurzschlüsse zu allen anderen Signalarten ausgeschlossen sind.

Autor: Josef Hübl

2. Lösungsansatz

Wurde zu einem gerichteten Graphen mit seinen gefärbten Ein- und Ausgängen ein Set von gefärbten Bäumen gefunden und schreitet man ausgehend von den Ausgängen (targets) entgegen der Kantenrichtung auf den Bäumen voran, so erhält man ein Set von fast knoten-disjunkten Pfaden, wobei höchstens der Endknoten eines Pfades auch Knoten eines anderen Pfades sein kann. Kurz gesprochen, erhält man also ein Set von knoten-disjunkten Pfaden, deren Anfangs und Endknoten dieselbe Farbe besitzen.

Ist umgekehrt ein Set knoten-disjunkter Pfade gegeben, die jedem Ausgang einen gleich gefärbten Eingang zuordnen, so bilden diese zusammen eine Lösung für das „Colored-Trees“-Problem.

Der Lösungsansatz für einen Algorithmus, der ein solches Set disjunkter Pfade finden kann, besteht darin, ausgehend von den Ausgangskonten (targets) solange entgegen der Richtung der Kanten rückwärts zu gehen und die Vorgänger mit der selben Farbe einzufärben, bis er auf einen bereits eingefärbten Knoten mit passender Farbe trifft. Da der Graph keine Zykel enthält, kann es sich bei einen solchen Knoten nur um einen Eingangsknoten (sources) oder einem bereits eingefärbten Knoten eines anderen Pfades handeln; nicht aber um einen Knoten des gleichen Pfades auf dem vorangegangen wird. Wird zudem bei der Suche so vorgegangen, dass zuerst ein Pfad für einen Ausgangsknoten abgeschlossen wird, bevor mit dem Pfad für einen weiteren Ausgangsknoten begonnen wird, so ist sichergestellt, dass von dem eingefärbten Knoten, auf den man gestoßen ist, bereits ein zielführender Pfad zu einem gleich gefärbten Eingangsknoten (sources) führt. (Unter Umständen kann dieser seinerseits über andere Pfade zum Ziel führen.)

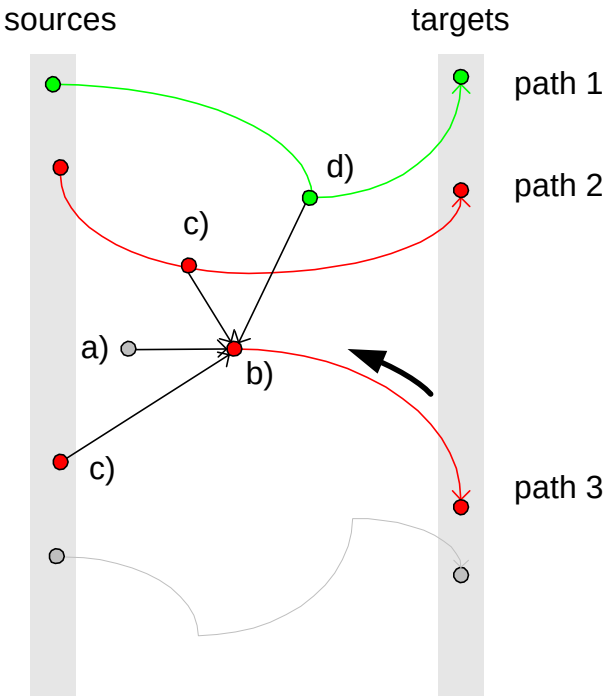
Prüft der Algorithmus nacheinander alle möglichen Kombinationen von disjunkten Pfaden, so wird entweder eine Lösung gefunden oder es ist sicher ausgeschlossen, dass eine Lösung existiert.

3. Der Algorithmus

3.1 Beschreibung

Die Eingangs- und Ausgangskonten werden als eingefärbt vorausgesetzt. Ebenso gegeben ist ein Array von Pfaden bestehend aus genau einem Ausgangsknoten. Der Algorithmus erweitert diese Pfade und bedient der sich dabei der sich rekursiv gegenseitig aufrufenden Routinen `ValidStep()` und `ValidPath()`.

`VaildStep()` versucht entsprechend der Tiefensuche (DFS) einen Pfad von einem Ausgangsknoten zu einem Eingangsknoten oder bereits anderen passend eingefärbten Knoten zu finden. Konnte ein solcher gefunden werden so wird mit `ValidPath()` geprüft, ob sich auch für alle noch verbliebenen Ausgangsknoten disjunkte Pfade finden lassen. Dazu ruft `ValidPath()` die Routine `ValidStep()` mit dem nächsten Ausgangsknoten als Startknoten.



Die möglichen Fälle von `ValidStep()`

Steht der Algorithmus im Fall von `ValidStep()` auf einem bestimmten Knoten x, so werden nacheinander die Kanten zu all seinen Vorgängern y geprüft. Dabei können 4 verschiedene Situationen auftreten können:

a) Der Vorgänger y ist noch nicht gefärbt. Er wird mit der Farbe des Ausgangsknoten eingefärbt, auf den Pfad kopiert und `ValidStep()` mit y als Startknoten aufgerufen.

b) Es gibt keinen bzw. keinen weiteren Vorgänger. D.h. der Pfad kann nicht zielführend vervollständigt werden und der Routine kehrt `false` zurück um ihre Suche ausgehend vom zuletzt geprüften Knoten (vorletzter Knoten auf dem Pfad) fortzusetzen.

c) Der Vorgänger y ist bereits gefärbt und zwar mit der passenden Farbe. D.h. der Pfad kann mit y abgeschlossen werden. Mit `ValidPath()` wird geprüft, ob auch für alle noch verblieben Ausgangsknoten ein zielführender Pfad gefunden werden kann. Ist dies der Fall so wurde eine Lösung gefunden. Ist dies nicht der Fall, so wird y verworfen und `ValidStep()` setzt seine Suche mit dem nächsten Vorgänger fort.

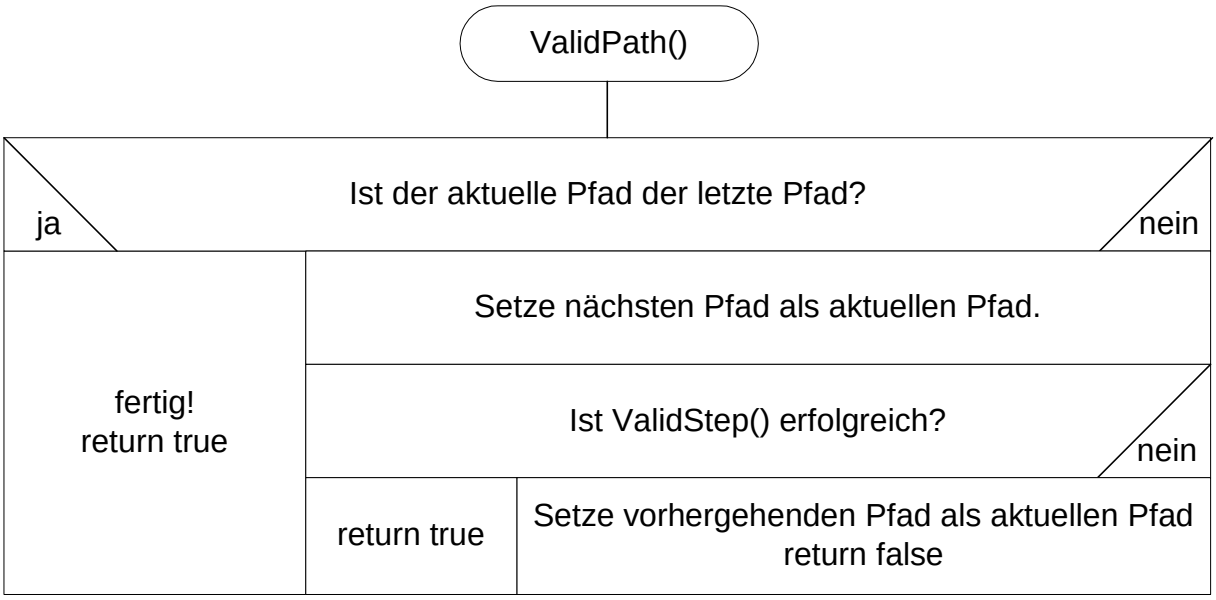
d) Ist der Vorgänger y bereits gefärbt, aber mit nicht passender Farbe, so wird er ignoriert und die Suche mit dem nächsten Vorgänger fortgesetzt.

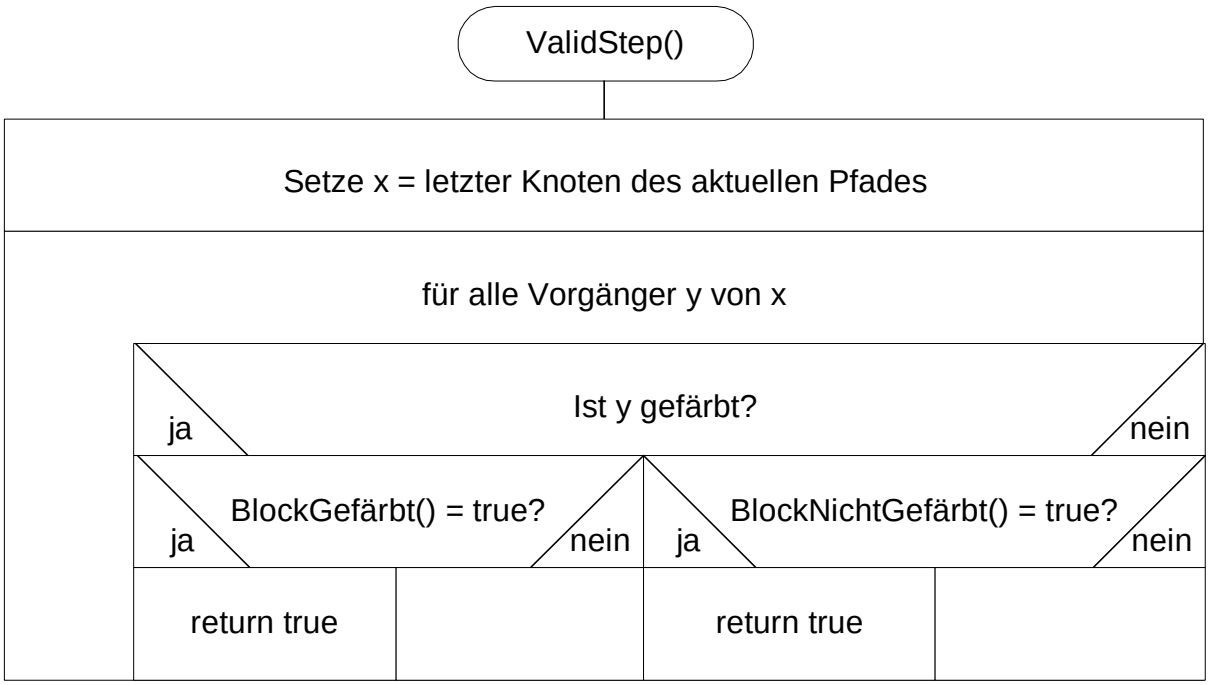
Da der Algorithmus der Reihe nach zu allen Ausgängen einen zielführenden Pfad sucht, kann sehr einfach über Induktion nachgewiesen werden, dass er korrekt arbeitet. Gibt es nur einen Ausgangsknoten, so finden `ValidStep()` durch rekursiven Aufruf seiner selbst einen zielführenden Pfad, sofern einer existiert, da nach Tiefensuche (DFS) vorgegangen wird. Damit ist der Induktionsanfang gegeben.

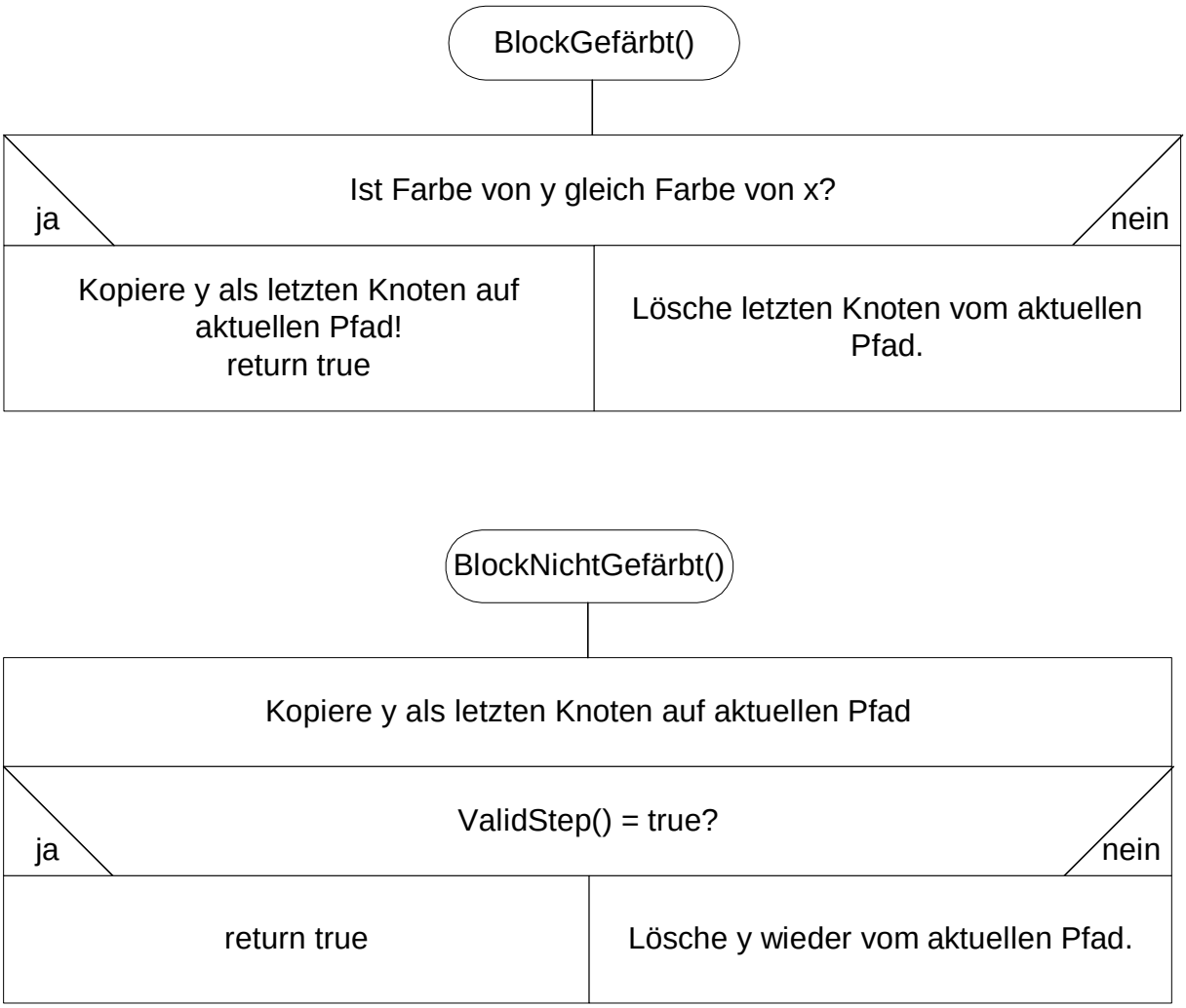
Angenommen der Algorithmus arbeitet für n Ausgangsknoten korrekt und es sind n+1 Ausgangsknoten gegeben. `ValidStep()` probiert nacheinander für alle vom ersten Ausgangsknoten ausgehenden zielführenden Pfade durch Aufruf von `ValidPath()`, ob für die verbleibenden n Ausgangsknoten, das Set von disjunkten Pfaden vervollständigt werden kann. Nach Induktionsvoraussetzung arbeitet der Algorithmus dabei korrekt. D.h. wenn es ein Set von n+1 disjunkten Pfaden gibt, so findet der Algorithmus dieses, da alle möglichen Kombinationen durchprobiert werden.

Autor: Josef Hübl

3.2 Flußdiagramme







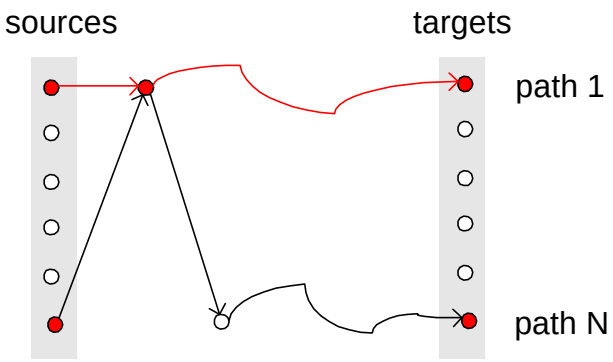
3.3 Aufwandsabschätzung

Es ist bekannt, dass das „Colored -Trees"-Problem NP-vollständig ist. D.h. der Aufwand eines jeden Algorithmus, der dieses Problem löst wächst exponentiell.

Bei dem vorliegenden Algorithmus prüft die Routine `ValidStep()` für jeden Knoten eines Pfades alle dessen Vorgänger. Somit ist der Algorithmus von $O(v^n)$, wobei v die maximale Anzahl von Vorgängern und n die Anzahl der Knoten ist.

Ein Beispiel für den „Worst-Case" läßt sich sehr einfach konstruieren. Dabei muss lediglich dafür gesorgt werden, dass der Pfad zum letzten Ausgangsknoten über einen Knoten laufen muss (weil es keine andere Kante gibt) den der Algorithmus gleich zu Beginn für den Pfad zum ersten Ausgangsknoten vereinnahmt. Der Algorithmus probiert dann für die dazwischen

liegenden Ausgangsknoten alle Kombinationen von Pfaden durch, bevor er den im ersten Pfad gemachten Fehler korrigiert.



Beispiel für den „Worst-Case“

Der Algorithmus kann noch etwas optimiert werden, in dem Eingangskonten (Ausgangskonten), die nur einen Nachfolger (Vorgänger) haben durch diesen ersetzt bzw. der Nachfolger (Vorgänger) passend eingefärbt wird, bevor der Algorithmus startet. In einer Nachbearbeitungsphase müssen die gefundenen Pfade dann entsprechend erweitert werden.

Auch können Untergraphen die nur aus einem Pfad bestehen zu einer einzigen Kante geschrumpft werden, die nach Beendigung des Algorithmus wieder expandiert werden muss, falls sie Bestandteil eines gefärbten Pfades ist.

Autor: Josef Hübl

4. Programmcode

```
#include "path.h"

class Router
{
public:
	Router( const Graph &g, NodeAttribute<int> &color, NodeAttribute<int> &id ) : m_graph(
g ), m_color( color ), m_id( id ) {};
	bool SearchColoredPathes( int numberOfPathes, NodePath *pathes );
	bool CheckPathes( int numberOfSourceNodes, Node *sourceNodes, int numberOfTargetNodes,
Node *targetNodes, NodePath *pathes );
	bool IsEdge( Node &source, Node &target); //später zu graph

private:
	bool ValidStep( void );
	bool ValidPath( void );

	const Graph &m_graph;
	NodeAttribute<int> &m_color;
	NodeAttribute<int> &m_id; // for debugging
	int m_noneColor;
	int m_numberOfPathes;
	int m_currentPathColor;
	int m_currentPathNumber;
	NodePath *m_currentPath;
};



bool Router::SearchColoredPathes(  int numberOfPathes, NodePath *pathes )
{
	m_numberOfPathes = numberOfPathes;
	m_currentPath = &(pathes[0]);
	m_currentPathNumber = 0;
	m_currentPathColor = m_color[ m_currentPath->First() ];
	m_noneColor = 0; // color.GetDefaultValue();

	return ValidStep();
}


bool Router::ValidPath( void )
{
	if ( m_currentPathNumber == (m_numberOfPathes - 1))
		return true; // no other path

	// try to finish all next pathes
	m_currentPath++;
	m_currentPathNumber++;
	m_currentPathColor = m_color[ m_currentPath->First() ];

	if ( !ValidStep() )
	{
		// no valid pathes found
		m_currentPath--;
		m_currentPathNumber--;
		m_currentPathColor = m_color[ m_currentPath->First() ];
		return false;
	}

	return true;
}
```


 Autor: Josef Hübl

```
bool Router::ValidStep( void )
{
	Node last = m_currentPath->Last();
	Edge edge = last.firstInEdge();
	Node pre;

	while (edge)  // for all predecessors of last
	{
		pre = edge.source();

		if( m_color[ pre ] != m_noneColor )
		{
			// source node is already colored
			if ( m_color[ pre ] == m_currentPathColor )
			{
				// path finished
				m_currentPath->AddLast( pre );
				if ( ValidPath() )
					return true;
				else
				{
					m_currentPath->DelLast( );
				}
			}
		}
		else
		{
			// source node is uncolored
			m_currentPath->AddLast( pre );
			m_color[ pre ] = m_currentPathColor;
			if ( ValidStep() )
				return true;
			else
			{
				m_currentPath->DelLast();
				m_color[ pre ] = m_noneColor;
			}

		}
		edge = edge.nextInEdge();
	}
	return false;
}
```


5. Varianten

5.1 Beschränkungen

Zur Lösung des Relationenproblems wird eine Variante dieses Algorithmus benötigt. Dabei dürfen sich die Pfade nicht auf beliebigen Knoten treffen, sondern nur auf speziell als „multible“ gekennzeichneten. Trifft der Suchalgorithmus auf einen gefärbten Konten, so muss dieser nicht nur die richtige Farbe besitzen, sondern auch noch als „multible“ gekennzeichnet sein um den Pfad abzuschließen. Im Verneinungsfall verhält sich der Algorithmus so, als wäre der aktuelle Knoten mit einer anderen, nicht passenden Farbe gekennzeichnet.

5.2 Optimierungen

5.2.1 Komponentenbasierte Optimierungen

Gibt es keine Lösung, so stellt der „Colored-Tree“-Algorithmus (in seiner einfachen Variante) dies erst nach dem Ausprobieren aller Pfad-Kombinationen fest. Da die Anzahl dieser Kombinationen mit Fakultät anwächst, bedeutet dies in der Praxis, dass der Algorithmus dies „nie“ feststellt, da er „nie“ zu Ende kommt.

Es gibt jedoch einige sehr einfache Methoden mit denen mit linearem oder quadratischem Aufwand festgestellt werden kann, ob überhaupt eine Lösung existieren kann, (ohne alle Kombinationen durch zu probieren.) da wesentliche Instrument dabei ist die Zusammenhangskomponente.

Jeder Graph zerfällt in maximale Teilgraphen, die untereinander weder durch Kanten noch Pfade und Wege verbunden sind. Diese heißen die Zusammenhangskomponenten des Graphen. Die Zusammenhangskomponenten können sehr einfache über ein DFS- oder BFS-Markierungsverfahren der Ordnung O(n) ermittelt werden.

Im Bezug auf den „Colored-Tree“-Algorithmus gelten folgende einfach verifizierbaren Gesetzmäßigkeiten:

1. Gibt es eine Zusammenhangskomponente, für die das „Colored-Tree“-Problem nicht gelöst werden kann, so kann es für den Gesamtgraphen ebenfalls nicht gelöst werden.

 Autor: Josef Hübl

2. Gibt es in einer Zusammenhangskomponente zu einem gefärbten Ausgangsknoten nicht mindestens einen identisch gefärbten Eingangsknoten, so kann das „Colored-Tree“-Problem weder für diese Komponente noch für den gesamten Graphen gelöst werden.
3. Gibt es in einer Zusammenhangskomponente zu einem gefärbten Ausgangsknoten keinen Pfad von einem identisch gefärbten Eingangsknoten, so kann das „Colored-Tree“-Problem weder für diese Komponente noch für den gesamten Graphen gelöst werden.

Die Gesetzmäßigkeit 3. kann einfach überprüft werden, indem dem beschriebenen „Colered-Trees“-Algorithmus eine Routine `FastCheck()` vorgeschaltet wird, die für jeden Ausgangsknoten getrennt mit `ValidStep()` versucht einen Pfad ausgehend von einem identisch gefärbten Eingangsknoten zu finden, wobei die betretenen Knoten nachträglich wieder entfärbt werden. Gelingt dies nicht, so gibt es mindestens eine Zusammenhangskomponente für die das „Colored-Tree“-Problem unlösbar ist. Da die Gesetzmäßigkeit 3. stärker als Gesetzmäßigkeit 2. ist, wird sie von dieser Routine gleich mit behandelt.

Die Gesetzmäßigkeit 1. kann man sich zu Nutze machen, wenn die gefärbten Ausgangsknoten als Endknoten der gesuchten Pfade vor Start des „Colered-Trees“-Algorithmus‘ entsprechend ihrer Zugehörigkeit zu den Zusammenhangskomponenten sortiert werden. Kehrt in diesem Fall die Routine `ValidPath()` mit `false` zurück und gehört der Endknoten des dann wieder aktuellen Pfades zu einer anderen Zusammenhangskomponente wie der zuletzt untersuchte, so wurden für die eben abgeschlossene Zusammenhangskomponente alle Pfad-Kombinationen ohne Erfolg erprobt. Der Algorithmus kann somit als Ganzes ohne Erfolg abgebrochen werden.

Es empfiehlt sich auch dabei die Zusammenhangskomponenten nach ihrer Größe d.h. Anzahl Knoten zu sortieren, damit kleinere Komponenten zuerst getestet werden und das zeitaufwendige aber nutzlose Testen großer Komponenten vermieden wird.

5.2.2 Graphzerstörende Optimierungen

Gibt es in einem Graphen ungefärbte Knoten, die keine Vorgänge oder Nachfolger besitzen, so können sie unmöglich auf einem Pfad von einem gefärbten Ein- zu einem gefärbten Ausgangsknoten liegen und somit aus dem Graphen gelöscht werden. Da für den dadurch reduzierten Graphen dieselbe Gesetzmäßigkeit gilt kann die Reduktion solange fortgesetzt werden bis jeder Knoten entweder gefärbt ist oder mindestens einen Vorgänger und einen Nachfolger besitzt.

Insbesondere bedeutet dies auch, dass ganze Pfade entfernt werden auf denen alle Knoten ungefärbt sind und genau einen Vor- und Nachfolger besitzen. In dünnbesiedelten Graphen

mit wenig gefärbten Ein- und Ausgangsknoten kann dies zu einer erheblichen Reduktion der Komplexität führen.

Der Nachteil dieser Optimierung ist jedoch der, dass der Graph teilweise zerstört wird und daher unter Umständen auf einer Kopie gearbeitet bzw. eine Kopie erstellt werden muss.

Autor: Josef Hübl