Ist die Bedingung nicht erfüllt, so wird $m_{i,j}$ auf 0 gesetzt. Die Refinement Procedure wird so oft wiederholt, bis keine Veränderung der Matrix mehr eintritt, und macht dann eine Überprüfung der Bedingung 3.3 überflüssig. Dadurch dass Einsen durch Nullen ersetzt werden, wird die Anzahl der Möglichkeiten in folgenden Iterationen verringert und, sofern Zeilen entstehen, die keine einzige 1 mehr aufweisen, kann umgehend ein Backtrackingschritt erfolgen. Die Wirkung der Refinemente Procedure soll verstärkt werden, indem die Zeilen der Matrix in einer Reihenfolge bearbeitet werden, die der Sortierung der zugehörigen Knoten nach absteigendem Grad entspricht.

Bei der Implementierung des Verfahrens müssen Vorkehrungen getroffen werden, um den Zustand der Matrix nach einem Backtrackingschritt wieder herstellen zu können. Für ein Suchmuster mit n Knoten und einen Graphen mit m Knoten wird in [51] die Worst-Case-Laufzeit des Algorithmus mit $O(m^n n^2)$ angegeben, der benötigte Speicherbedarf kann durch $O(m^3)$ abgeschätzt werden [52].

3.2.3 VF2

Der VF2-Algorithmus basiert genau wie der Algorithmus von Ullmann auf Backtracking. Es wird wiederum versucht, eine partielle Abbildung schrittweise auszubauen, wobei jetzt jedoch stets benachbarte Knoten hinzugefügt werden. Durch zusätzliche Bedingungen, die sich mit geringem Aufwand auswerten lassen, soll dabei der Suchraum beschränkt werden. Der Algorithmus wurde 1999 von Vento et al. vorgestellt [20, 18] und nachfolgend verbessert [19, 52]. Durchgeführte Vergleichsstudien zeigen, dass der VF2-Algorithmus eine deutlich bessere Laufzeit als der Algorithmus von Ullmann aufweist [30]. Mit der Bibliothek *vflib* steht eine freie Implementierung des Algorithmus zur Verfügung. Während die Bibliothek verschiedene Varianten des Graph-Matchings unterstützt, beziehen sich die Veröffentlichungen in erster Linie auf GI und ITGI[3].

Der VF2 Algorithmus beschreibt den Suchraum durch Zustände, wobei jedem Zustand s eine partielle Abbildung $M(s) \subset V_1 \times V_2$ zugeordnet wird. Initial wird ein Zustand s_0 erzeugt mit $M(s_0) = \emptyset$. Ein Zustand wird in einen neuen überführt, indem ein Paar (p, h) mit $p \in V_1$ und $h \in V_2$ der Abbildung hinzugefügt wird. Für alle weiteren Zustände, die aus diesem entstehen, gilt also, dass der Knoten p auf h abgebildet wird. Dabei werden nur Zustände generiert, deren partielle Abbildung der geforderten Matching-Bedingung genügt. Ferner werden Zustände vermieden, deren partielle Abbildung zwar nicht direkt die Bedingung verletzt, für die aber ausgeschlossen werden kann, dass sie sich zu einer zulässigen Lösung erweitern lassen (*Pruning*).

Der Pseudocode 1 wurde [52] entnommen und beschreibt die rekursiv verwendete Funktion MATCH. Wird ein Zustand erreicht, in dem alle Knoten aus V_1 auf einen Knoten von

[3] In den Veröffentlichungen zu VF2 wird eine abweichende Definition verwendeten: Was dort als "graph-subgraph isomorphism" bezeichnet wird, entspricht der hier verwendeten Definition des ITGI. Die Bibliothek bietet TGI unter der Bezeichnung "monomorphism", vgl. Abschnitt 3.1.1.

Algorithmus 3.1 : MATCH(s)-Funktion des VF2-Algorithmus.

Eingabe : Zustand s mit Graphen $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ und partieller Abbildung $M(s)$; initial s_0 mit $M(s_0) = \emptyset$

Ausgabe : Abbildung $M(s)$ von G_1 auf G_2

```
1 if M(s) enthält alle Knoten von G1 then
2     Gib M(s) aus                                   ▷ Abbildung gefunden
3 else
4     P(s) ← generateCandidatePairs(s)
5     forall p ∈ P(s) do
6         if isFeasible(p,s) then
7             s' ← Zustand, der sich durch Hinzufügen von p zu M(s) ergibt
8             match(s')
9     Datenstruktur wiederherstellen
```

G_2 abgebildet werden, wurde eine zulässige Lösung gefunden (Zeile 2). Ansonsten wird versucht, die Abbildung zu vervollständigen. Wird ein geeignetes Paar zur Erweiterung gefunden, erfolgt ein rekursiver Aufruf der MATCH-Funktion mit dem Zustand, der sich durch Hinzufügen des Paares ergibt (Zeile 8). Ansonsten erfolgt ein Backtrackingschritt, wobei zuvor Änderungen an der Datenstruktur rückgängig gemacht werden müssen (Zeile 9).

Die Funktionen generateCandidatePairs und isFeasible stellen sicher, dass jede zulässige Lösung gefunden werden kann und erfolglose Teilbäume frühzeitig als solche erkannt werden. Dazu werden *Terminalmengen* verwendet: Für einen Zustand s ist $T_1^{in}(s) \subset V_1$ die Menge der Knoten, die noch nicht in die partielle Abbildung $M(s)$ aufgenommen wurden, aber mit einem Knoten in $M(s)$ über eine ausgehende Kanten verbunden sind. $T_1^{out}(s)$ ist analog für Knoten definiert, die über eine eingehende Kante verbunden sind. $T_2^{in}(s)$ und $T_2^{out}(s)$ sind die Terminalmengen für G_2. Die Menge der Knoten, die nicht unmittelbar mit der partiellen Abbildung verbunden sind, wird als $N_1(s)$ bzw. $N_2(s)$ bezeichnet. Abbildung 3.4 verdeutlicht den Zusammenhang für einen Zustand, in dem bereits vier Knoten eines Graphen abgebildet wurden.

Die Funktionen generateCandidatePairs findet neue Kandidaten, die der Abbildung hinzugefügt werden können. Dazu werden zunächst die Terminalmenge beider Graphen betrachtet, so dass die Abbildung möglichst um benachbarte Knoten erweitert wird. Falls $T_1^{out}(s)$ nicht leer ist, so ist $P(s) = \{min(T_1^{out}(s))\} \times T_2^{out}(s)$, wobei $min(T_1^{out}(s))$ das minimale Element aus $T_1^{out}(s)$ bezüglich einer beliebigen, festen totalen Ordnung meint. Sonst wird $P(s)$ analog aus $T_1^{in}(s)$ und $T_2^{in}(s)$ gebildet. Sind beide Terminalmengen leer (der Graph ist also unzusammenhängend), so wird $P(s)$ analog aus den Knoten außerhalb von $M(s)$ bestimmt. Aufgrund der totalen Ordnung steht in jedem Zustand fest, welcher

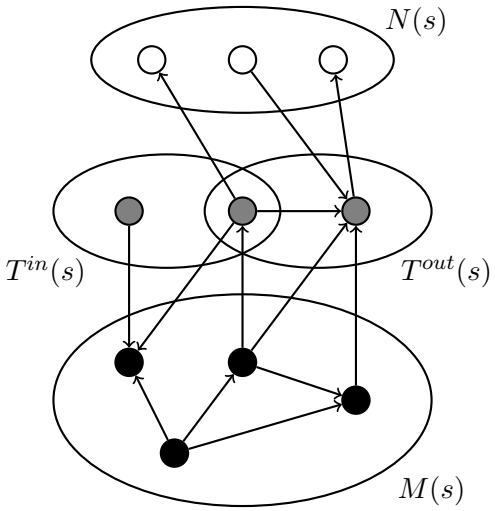


Abbildung 3.4: Die Knoten in der partiellen Abbildung $M(s)$ sind schwarz dargestellt, die daraus über eine Kante erreichbaren Knoten in den Terminalmengen $T^{in}(s)$ bzw. $T^{out}(s)$ grau. Nicht direkt verbundene Knoten der Menge $N(s)$ sind weiß dargestellt.

Knoten aus V_1 hinzugefügt wird, wodurch vermieden wird, dass eine partielle Abbildung über unterschiedliche Wege mehrmals erzeugt wird.

Die Funktion isFeasible prüft, ob das Hinzufügen des Paares die Matching-Bedingung verletzt und prüft mittels einer "k-Look-Ahead"-Regel, ob nach k Schritten noch ein konsistenter Folgezustand existieren kann. Die Implementierung unterscheidet sich je nach Matching-Variante. Für den Fall von ITGI werden Regeln aufgestellt, die mittels Terminalmengen einen 2-Look-Ahead für ein Paar (p, h) erreichen:

0-Look-Ahead: Für alle Kanten, die von $p \in V_1$ zu einem Knoten p' in der partiellen Abbildung führen, existiert eine entsprechende Kante von h zu einem h' mit $(p', h') \in M(s)$ und umgekehrt.

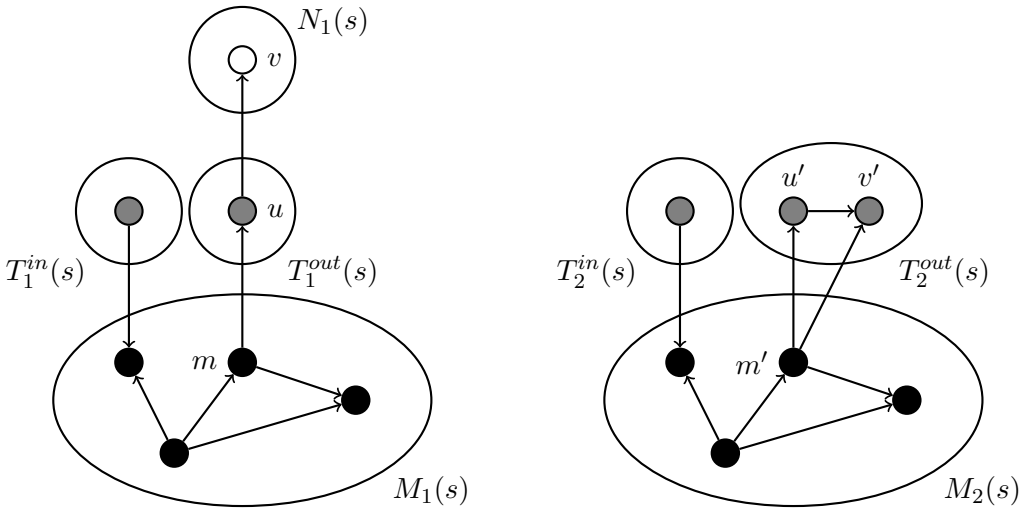
1-Look-Ahead: Die Anzahl der Nachbarn von p in T_1^{in} ist kleiner gleich der Anzahl der Nachbarn von h in T_2^{in}. Gleiches muss für die Anzahl der Nachbarn in T_1^{out} und T_2^{out} gelten.

2-Look-Ahead: Die Anzahl der Nachbarn von p in $N_1(s)$ ist kleiner gleich der Anzahl der Nachbarn von h in $N_2(s)$.

Außerdem wird geprüft, ob Knoten- und Kantenlabel zueinander passen.

Teilgraph-Isomorphie Für das TGI-Problem müssen die Look-Ahead-Regeln der Funktion isFeasible angepasst werden. Dem Quellcode der vflib lassen sich folgende Änderungen gegenüber den Regeln für ITGI entnehmen:

0-Look-Ahead: Der umgekehrte Fall wird nicht mehr geprüft.



(a) G_1 (b) G_2

Abbildung 3.5: Von dem Graphen G_1 (a) lässt sich ausgehend vom Zustand s ein TGI auf G_2 (b) ableiten, wobei u auf u' und v auf v' abgebildet werden. Da G_2 eine Kante (m', v') enthält, während eine Kante (m, v) in G_1 nicht existiert, befinden sich v und v' in unterschiedlichen Mengen.

1-Look-Ahead: Entspricht der Regel für ITGI.

2-Look-Ahead: Sei n_1 die Anzahl der Nachbarn von p in $N_1(s)$, i_1 in $T_1^{in}(s)$ und o_1 in $T_1^{out}(s)$, sowie n_2, i_2, o_2 analog für h. Es muss gelten $n_1 + i_1 + o_1 \leq n_2 + i_2 + o_2$.

Die 2-Look-Ahead-Regel muss also abgeschwächt werden, damit keine zulässigen Lösungen ausgeschlossen werden. Im Fall von ITGI konnten Knoten aus $T_1^{in}(s)$, $T_1^{out}(s)$ und $N_1(s)$ nur auf Knoten der entsprechenden Menge von G_2 abgebildet werden. Abbildung 3.5 verdeutlicht, dass dieser Zusammenhang für einen TGI nicht gilt, da Knoten aus $N_1(s)$ jetzt auf Knoten aus $T_2(s)$ abgebildet werden können.

Implementierung Neben der partiellen Abbildung $M(s)$ werden mit dem Zustand die Terminalmengen für G_1 und G_2 verwaltet. Dazu werden sechs Felder verwendet: `core_1` und `core_2` haben die Dimension von $|V_1|$ bzw. $|V_2|$ und speichern die Abbildung. Wenn $p \in V_1$ auf $h \in V_2$ abgebildet wird, ist `core_1[p]=h` und `core_2[h]=p`, wobei die Knoten durch ihre Indizes identifiziert werden. Die vier Terminalmengen werden ebenfalls mit Feldern gespeichert.

Wird ein Paar der Abbildung hinzugefügt, müssen die Abbildung und die Terminalmengen entsprechend angepasst werden. Um den Speicherbedarf gering zu halten, wird bei dem rekursiven Aufruf der MATCH-Funktion kein komplett neuer Zustand erzeugt, sondern alle Zustände teilen sich die sechs Felder. Dadurch wird es erforderlich, dass beim Backtracking Änderungen rückgängig gemacht werden (Algorithmus 3.3, Zeile 9). Um genau die Knoten aus den Terminalmengen entfernen zu können, die im Schritt zuvor eingefügt

worden sind, wird in den Feldern die Tiefe des Suchbaums (bzw. Größe der partiellen Abbildung) gespeichert, bei der der Knoten in die Terminalmenge aufgenommen worden ist. Somit wird nur wenig Speicher für einen Zustand und rekursiven Aufruf benötigt.

Für die Laufzeit des Algorithmus wird $O(m!m)$ angegeben und der Speicherbedarf durch $O(m)$ abgeschätzt, wobei $m = |V2| \geq |V_1|$ ist. Aufgrund des geringen Speicherbedarfs eignet sich der VF2-Algorithmus auch für sehr große Graphen.

3.2.4 Suchplanoptimierung

Backtracking-Algorithmen für das TGI-Problem erweitern eine partielle Abbildung meist schrittweise, indem Elemente (Knoten und Kanten) des Suchmusters auf passende Elemente des Zielgraphen so abgebildet werden, dass die TGI-Bedingung erfüllt bleibt. Dabei kann besonders für gelabelte Graphen die Reihenfolge, in der Elemente des Suchmusters in die Abbildung aufgenommen werden, einen starken Einfluss auf die Größe des Suchraums und somit auf die Laufzeit haben. Unter dem Begriff Suchplanoptimierung können Ansätze zusammengefasst werden, die durch vorherige Analyse der Graphen versuchen, eine möglichst günstige Reihenfolge zu finden. Heuristiken, um eine günstige Reihenfolge zu ermitteln, wurde vor allem im Bereich der *Graph Transformation* untersucht.

Batz [8] formuliert *primitive matching operations* für einen gerichteten gelabelten Graphen und definiert einen gültigen Suchplan als Folge solcher Operationen, die, der Reihe nach angewandt, das gesamte Suchmuster abdecken. Als *lookup operation* wird eine Operation bezeichnet, die ein Element des Suchmusters, das nicht direkt mit einem bereits abgebildeten Element verbunden ist, auf den Zielgraphen abgebildet wird. Eine *extension operation* beschreibt die Erweiterung der partiellen Abbildung um einen adjazenten Knoten und der zugehörigen Kante. Ziel ist es, einen Suchplan zu finden, bei dem für die Matchingoperationen möglichst wenige Auswahlmöglichkeiten bestehen. Dazu werden den Operationen Kosten in Höhe der durchschnittlichen Anzahl Kandidaten im Zielgraphen zugeordnet, wozu dieser zuvor analysiert werden muss. Mit Hilfe eines *Plan Graphs* wird anschließend ein Suchplan ermittelt, bei dem günstige Operationen möglichst früh ausgeführt werden und unvermeidbare, teure Operationen so spät wie möglich. Besonders für Graphen mit geringer Anzahl Kanten und einer Vielzahl unterschiedlicher Label verspricht die Methode gute Ergebnisse.

Andere Modelle wurden zuvor von Dörr und Zünndorf im Zusammenhang mit Graph Rewriting Tools vorgestellt. Dörrs [23] Ansatz ermöglicht es, das TGI-Problem für gelabelte Graphen in Linearzeit zu lösen, wenn ein Suchplan mit Matchingoperationen ohne Auswahlmöglichkeiten existiert. Zünndorf beschreibt eine heuristische Optimierung des Suchplans für das Graph Rewriting Tool Progres [78].