Datenstrukturen und Algorithmen, SS 2024
Prof. Dr. P. Rossmanith
D. Cloerkes, M. Gehnen, D. Mock, N. Speetzen

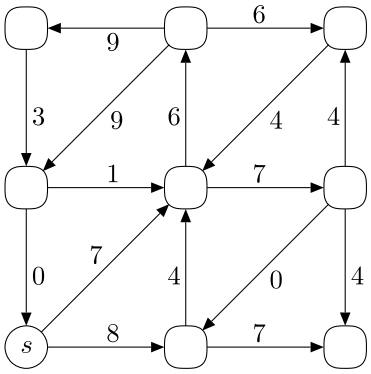
Abgabe: 14.06.2024

Übungsblatt mit Lösungen 08

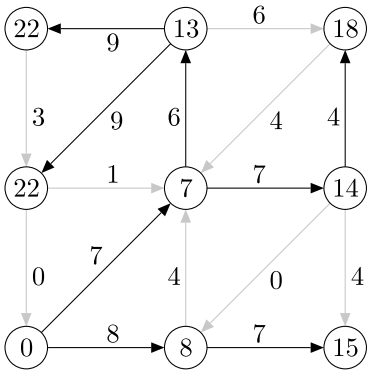
Aufgabe T27

Finden Sie die kürzesten Wege zu allen Knoten vom Startknoten s, indem Sie den Algorithmus von Dijkstra verwenden.

Geben Sie dazu die resultierenden Distanzen von s zu allen Knoten an, indem Sie diese in die Knoten des Graphen eintragen. Markieren Sie außerdem jede Kante, die zu einem kürzesten Weg von s aus gehört.

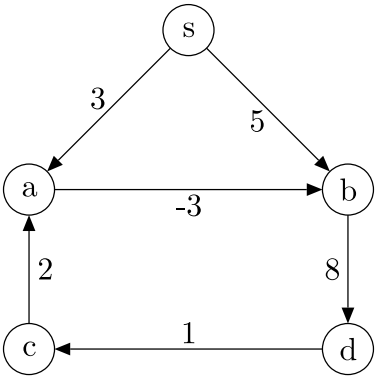


Lösungsvorschlag



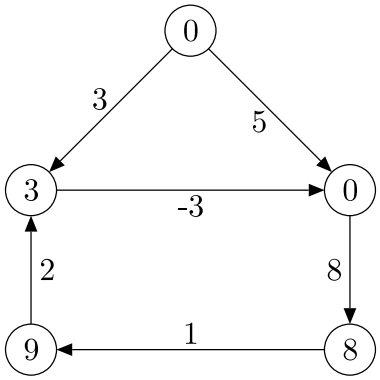
Aufgabe T28

Wenden Sie den Algorithmus von Bellman und Ford auf folgendes Netzwerk an. Verwenden Sie s als Startknoten.



Lösungsvorschlag

Es ergeben sich folgende Abstände:



Aufgabe T29

Beweisen Sie den zweiten Punkt von Lemma A: $f(X,Y) = -f(Y,X)$ für $X, Y \subseteq V$, falls f ein Fluss für $G = (V, E)$ ist.

Lösungsvorschlag

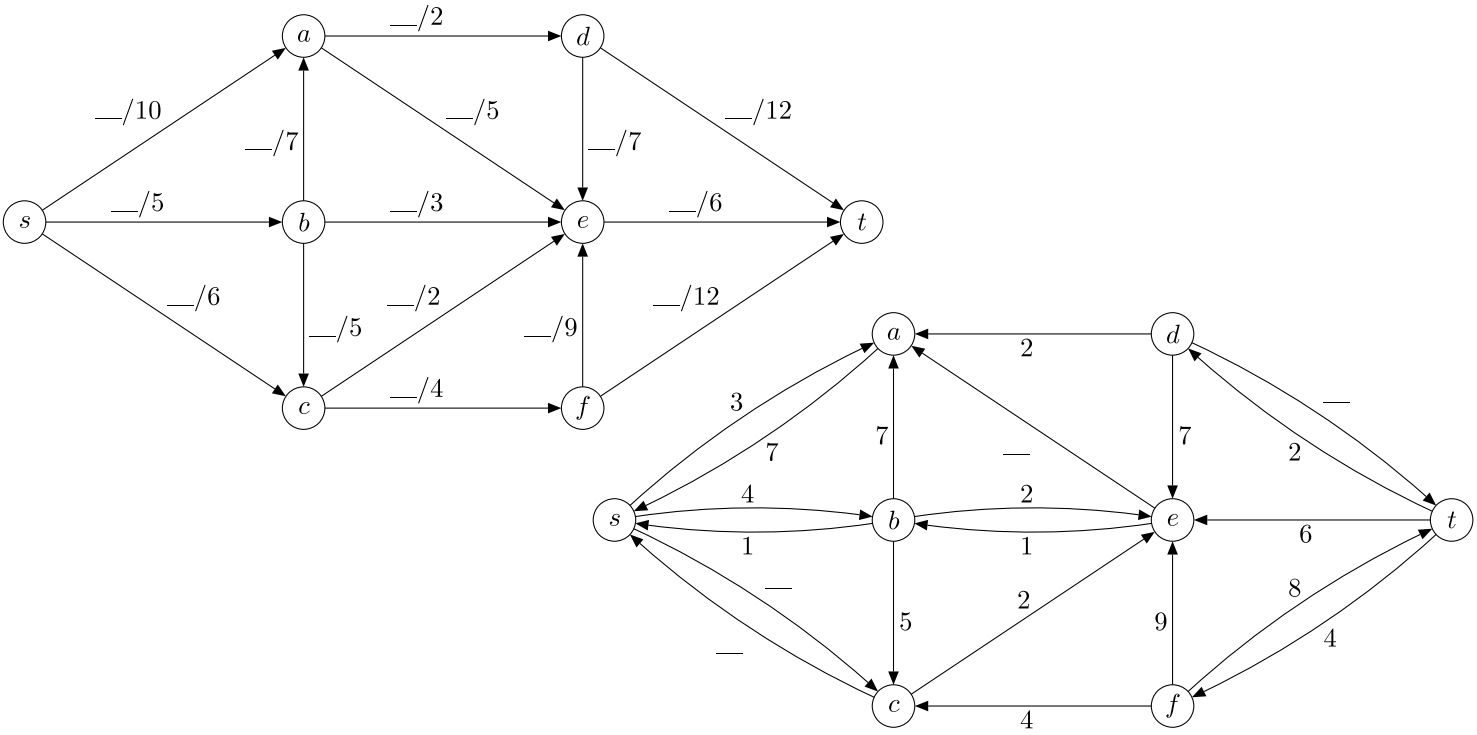
$$f(X,Y) \overset{Def.}{=} \sum_{x \in X} \sum_{y \in Y} f(x,y) \overset{Symm.}{=} \sum_{x \in X} \sum_{y \in Y} -f(y,x) =$$

$$\sum_{y \in Y} \sum_{x \in X} -f(y,x) = -\sum_{y \in Y} \sum_{x \in X} f(y,x) \overset{Def.}{=} -f(Y,X)$$

Aufgepasst: Welche Rolle spielen hier das Kommutativitäts-, das Assoziativ- und das Distributivgesetz?

Aufgabe T30

a) Sie finden links ein Flussnetzwerk G. Rechts ist das dazugehörige Residualnetzwerk G_f bezüglich eines Flusses f. Zeichnen Sie den Fluss f in die linke Zeichnung auf den Strichen ein und rechts die fehlenden Kapazitäten in G_f.

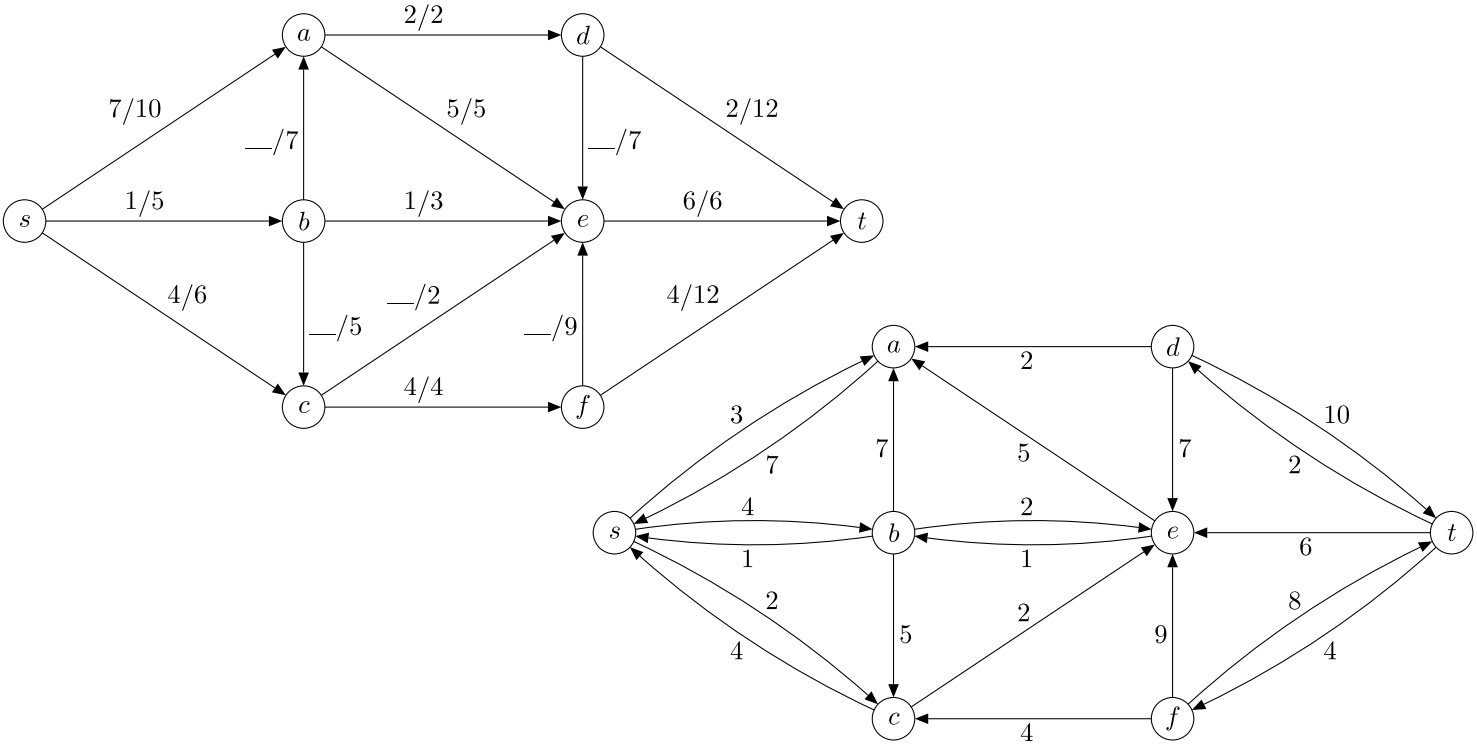


b) Berechnen sie den Wert des Flusses f.

c) Ist f maximal? Begründen Sie Ihre Antwort kurz.

Lösungsvorschlag

a)



b) Der Wert des Flusses f beträgt $|f| = 12$.

c) Ja, denn es gibt keinen s-t-Pfad im Residualnetzwerk.

Alternativ: Der gefundene Fluss ist maximal, da der Schnitt $(S, T) = (\{s, a, b, c, e\}, \{d, f, t\})$ die Kapazität 12 hat und es somit keinen größeren Fluss geben kann.

Aufgabe H26 (12 Punkte)

Der Algorithmus von Dijkstra kann auf gerichteten Graphen mit negativen gewichteten Kanten nicht verwendet werden. Das ist schade, da der Algorithmus eine deutlich geringere Zeitkomplexität als der allgemeinere Bellman-Ford Algorithmus hat.

Überlegen Sie sich, wie es möglich ist, mit gleicher Zeitkomplexität wie der des Algorithmus von Dijkstra kürzeste Pfade zwischen zwei Knoten s und t zu berechnen, sollte der gerichtete Graph genau eine negativ gewichtete Kante enthalten.

Lösungsvorschlag

In dem Graphen, welchem die negative Kante entnommen wurde, suchen wir zunächst die kürzesten Pfade vom Startknoten s zu allen anderen Knoten. Zu beachten ist, dass hiermit auch der Pfad zum Anfangsknoten der negativen Kante bekannst ist. Anschließend führen wir Dijkstra erneut aus, starten aber bei dem Endknoten der negativen Kante. Jetzt können wir vergleichen, ob der direkte Pfad s zu t ohne das Verwenden der negativen Kante kürzer ist, als der Pfad von s zum Startknoten der Kante, entlang der Kante selbst und vom Endknoten der Kante zum Zielknoten t.

Da wir Dijkstra nur zweimal ausführen, bleiben wir in derselben Zeitkomplexität wie Dijkstra selbst.

Aufgabe H27 (18 Punkte)

Der Alien Glorbo ist bei seinen Weltraumreisen auf ein Problem gestoßen: Er hat sich zu viel Wackelpudding gekauft. Um diesen nicht zurücklassen zu müssen, scheint die einzige Möglichkeit zu sein, einen seiner Treibstofftanks vollständig zu entleeren, um den Wackelpudding im Tank transportieren zu können.

Die Statusanzeige zeigt Glorbo die momentanen Füllmengen $x_i \in \mathbb{N}^+$ seiner drei (für unsere Zwecke unbegrenzt großen) Tanks. Glorbos Ziel ist, den Treibstoff so umzufüllen, dass ein Tank leer ist. Leider hat er hierbei ein paar Einschränkungen zu beachten:

- Er kann in einem Schritt immer nur ganzzahlige Litermengen auf einmal umfüllen, nie fraktionale Anteile.
- Er kann in einem Schritt immer nur von einem Tank i nach j umfüllen, wenn $x_i \geq x_j$.
- Damit der Druck im Zieltank nicht zu schnell ansteigt, wird immer genau soviel umgefüllt, dass sich die Füllmenge im Zieltank pro Schritt verdoppelt. Somit ist die Menge, die in einem Schritt umgefüllt wird immer $\min(x_i, x_j) = x_j$.

Da jeder Umfüllschritt unabhängig von der umgefüllten Menge sehr lange dauert, möchte Glorbo natürlich möglichst wenig Schritte durchführen.

Um Ihnen die Problemdefinition klarer zu machen, hat Glorbo sich bereits ein Beispiel überlegt: $x_0 = 5$ l, $x_1 = 17$ l und $x_2 = 42$ l. In diesem Beispiel sind die einzigen möglichen Umfüllschritte zu Beginn:

- $1 \to 0$: 10 l, 12 l, 42 l

- $2 \to 0$: 10 l, 17 l, 37 l
- $2 \to 1$: 5 l, 34 l, 25 l

Eine optimale Lösung hat hier z.B. vier Schritte: $1 \to 0, 2 \to 0, 2 \to 0, 1 \to 2$.

Nun liest Glorbo seine tatsächlichen Füllstände ab, welche zufälligerweise zu Teilen mit der kleinsten Matrikelnummer Ihrer Gruppe übereinstimmen. Von dieser Matrikelnummer bilden die ersten drei Ziffern x_0 und die letzten drei x_1 (z.B. "345678"$\to$ $x_0 = 345$ l, $x_1 = 678$ l). Unabhängig davon ist $x_2 = 1234$ l. Helfen Sie Glorbo!

Sein Bordcomputer beherrscht die Programmiersprachen Python, Java, C++ und Rust. Stellen Sie sicher, dass Sie Glorbo sowohl die optimalen Umfüllschritte, als auch ihren Programmcode zukommen lassen! (Bitte den Code ebenfalls in moodle hochladen.) Das Programm soll natürlich auf beliebigen Eingaben funktionieren, damit Glorbo es ggfs. wiederverwenden kann. Beschreiben Sie auch in Worten, wie Ihr Programm funktioniert.

Lösungsvorschlag

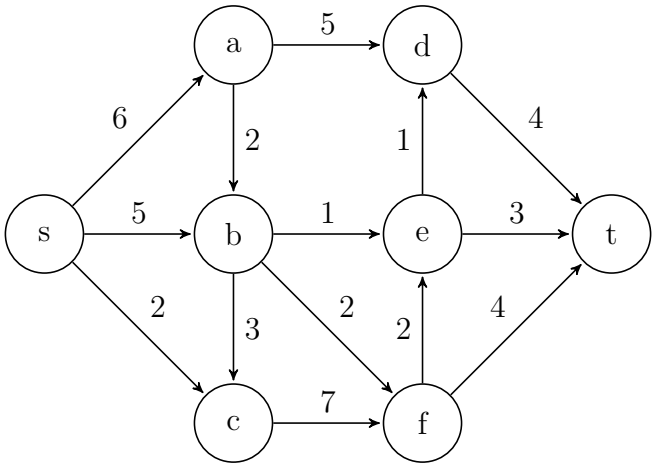
Das Problem kann mit Breitensuche gelöst werden. Jeder Schritt generiert neue Nachbarknoten, indem alle möglichen Umfüllschritte durchgeführt und in die Queue eingefügt werden. Sobald ein Knoten erreicht wird, bei dem ein $x_i = 0$ ist, kann terminiert werden.

Ein Beispiel: (der letzte Schritt kann in beide Richtungen durchgeführt werden):

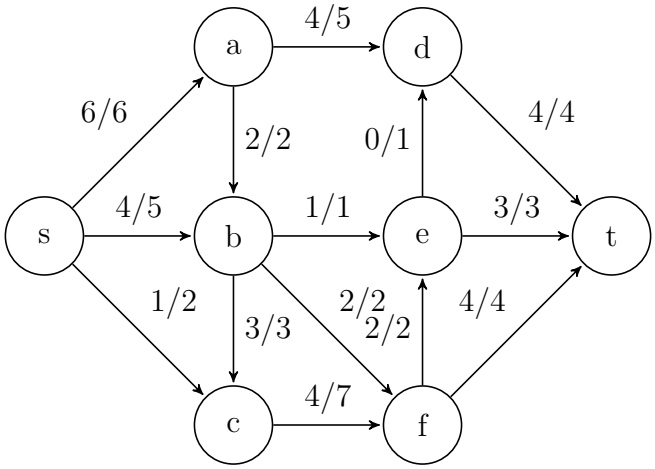
Initial state: 111 192 430 Move from 1 to 0 222 81 430 Move from 2 to 1 222 162 349 Move from 0 to 1 60 324 349 Move from 1 to 0 120 264 349 Move from 1 to 0 240 144 349 Move from 0 to 1 96 288 349 Move from 1 to 0 192 192 349 Move from 0 to 1 0 384 349

```
#include <chrono>
#include <cstdio>
#include <cstdlib>
#include <vector>

// declaration

struct state;
void print(state const&s, bool print_history);
void print_history_applied(state const&initial_state, std :: vector < std :: pair<int, int> > const&swap_history);
bool can_move_water(state const&s, int from_idx, int to_idx);
state move_water(state s, int from_idx, int to_idx);
bool has_empty_bucket(state const&s);
state find_move_order_bfs(state const&initial_state);


// implementation

struct state
{
    std :: vector<int> buckets;
    std :: vector < std :: pair<int, int> > move_history;
};

void print(state const&s, bool print_history = true)
{
    for(auto const&e : s.buckets)
    {
        printf("%d ", e);
    }
    if(print_history)
    {
        for(auto const&e : s.move_history)
        {
            printf("(%d, %d) ", e.first, e.second);
        }
    }
    printf("\n");
}

void print_history_applied(state const&initial_state, std :: vector < std :: pair<int, int> > const&swap_history)
{
    printf("Initial state: ");
    print(initial_state, false);
    state s = initial_state;
    for(auto move : swap_history)
    {
        printf("Move from %d to %d\n", move.first, move.second);
        s = move_water(s, move.first, move.second);
        print(s, false);
    }
}


bool can_move_water(state const&s, int from_idx, int to_idx) { return s.buckets[to_idx] <= s.buckets[from_idx]; }

state move_water(state s, int from_idx, int to_idx)
{
    int max_move_amount = s.buckets[to_idx];
    int move_amount = std :: min(max_move_amount, s.buckets[from_idx]);
    s.buckets[from_idx] -= move_amount;
    s.buckets[to_idx] += move_amount;
    s.move_history.push_back(std :: make_pair(from_idx, to_idx));
    return s;
}

bool has_empty_bucket(state const&s)
{
    for(auto const&e : s.buckets)
    {
        if(e == 0)
        {
            return true;
        }
    }
    return false;
}

state find_move_order_bfs(state const&initial_state)
{
    int n_nodes = 0;
    std :: vector<state> queue;
    queue.reserve(100000);
    queue.push_back(initial_state);
    while(!queue.empty())
    {
        auto state = queue.front();
        n_nodes++;
        if(n_nodes%10000 == 0)
        {
            printf("Visited %d nodes\n", n_nodes);
            fflush(stdout);
        }

        queue.erase(queue.begin());
        if(has_empty_bucket(state))
        {
            printf("Found solution after %d nodes\n", n_nodes);
            return state;
        }
        for(int i = 0; i < state.buckets.size(); i++)
        {
            for(int j = 0; j < state.buckets.size(); j++)
            {
                if(i != j && can_move_water(state, i, j))
                {
                    auto new_state = move_water(state, i, j);
                    queue.push_back(new_state);
                }
            }
        }
    }
    printf("No solution found\n");
    return initial_state;
}

int main(int argc, char * argv[])
{
    // read initial state
    int n = argc - 1;
    state initial_state;
    for(int i = 0; i < n; i++)
    {
        initial_state.buckets.push_back(atoi(argv[i + 1]));
    }

    // BFS
    auto start_time = std :: chrono :: high_resolution_clock :: now();
    state solution = find_move_order_bfs(initial_state);
    auto end_time = std :: chrono :: high_resolution_clock :: now();
    auto duration = std :: chrono :: duration_cast < std :: chrono :: milliseconds > (end_time - start_time).count();
    printf("Time taken: %ld ms\n", duration);

    // print path to solution
    print_history_applied(initial_state, solution.move_history);

    return 0;
}
```

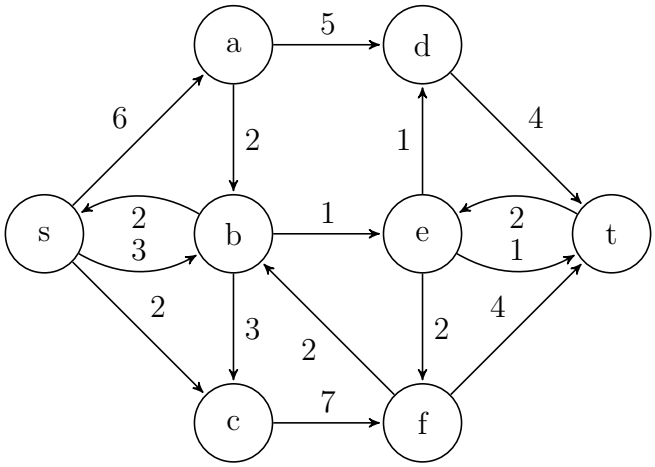
Aufgabe H28 (10 Punkte)
Wenden Sie die Ford–Fulkerson-Methode auf das folgende Flussnetzwerk an. Zeichnen Sie das resultierende Residualnetzwerk nach der ersten und letzten Augmentierung.



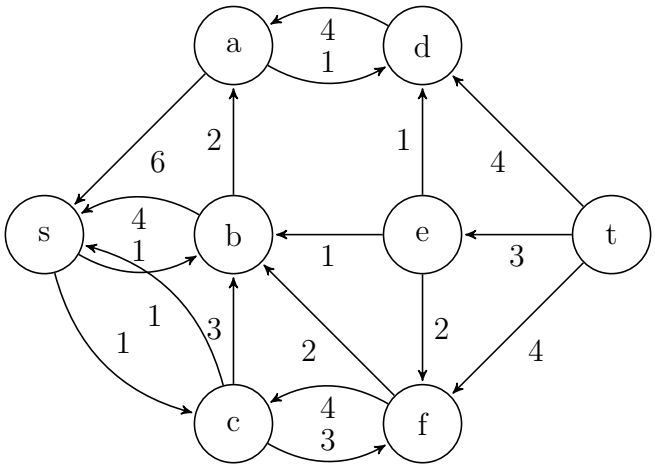
Lösungsvorschlag
Maximaler Fluss:



Residualnetzwerk nach der ersten Augmentierung (viele Möglichkeiten):



Residualnetzwerk nach der letzten Augmentierung:


a
d
s
b
e
t
c
f
4
1
6
2
1
4
4
1
1
3
1
1
3
2
2
4
4
3