Formalising Combinatorial Optimisation in Isabelle/HOL: Network Flows

Thomas Ammer

February 6, 2025

About Myself

- 2nd year PhD student at King's College London

Outline

Introduction and Aims

Network Flows

Mincost Flow Algorithms

Orlin's Algorithm

Formalisation Methodology

Running Time of Orlin's Algorithm

Limitations of Orlin's Algorithm

Summary

Table of Contents

Introduction and Aims

Network Flows

Mincost Flow Algorithms

Orlin's Algorithm

Formalisation Methodology

Running Time of Orlin's Algorithm

Limitations of Orlin's Algorithm

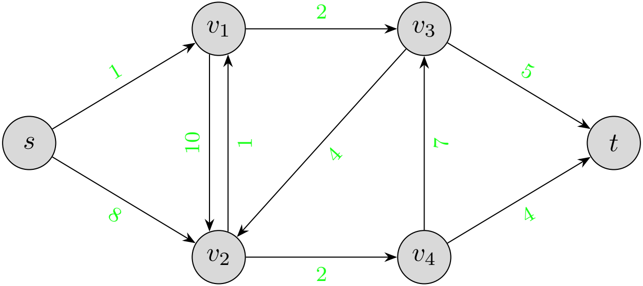
Summary

This Talk

- advanced Combinatorial Optimisation (CO)
- in the Isabelle/HOL prover
- mathematics where we aim to find an optimum solution for a problem that is based on a finite structure, e.g. graph
- simple examples: shortest path or spanning tree

Network Flows: Maximum Flows

- a directed Graph (V, E).
- find $f : E \to \mathbb{R}_0^+$
- edge capacities u: $f(e) \leq u(e)$
- ingoing flow = outgoing flow, two designated vertices s and t
- send as much flow as possible from s to t.

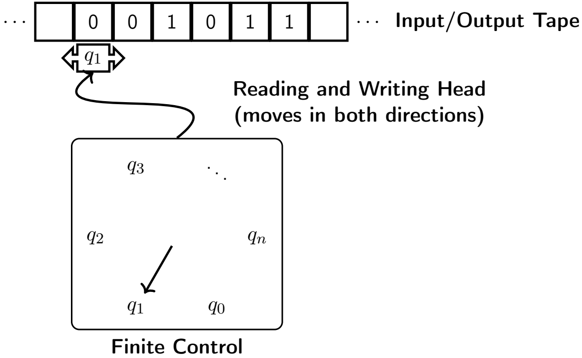


latex code taken from

`https://tex.stackexchange.com/questions/341949/how-to-plot-a-network-flow-with-tikz` **(modified)**

Algorithms and Running Time

- underlying structure finite $\Rightarrow$ compute a solution
- model of computation e.g. Turing Machine



latex taken from `https://texample.net/turing-machine-2/`

- running time = number of steps w.r.t. computation model
- running time as a term $t(n)$ depending on input size n
- polynomial and non-polynomial time algorithms
- good (= polynomial) running time is reason to study complicated algorithms
- brute force/enumeration (simple) vs. exploiting structure (complicated)

Aims of this Work and Context

- part of a bigger project (together with others): build a library of CO formalisations
- mathematics: graduate and research-level theory+algorithms
- pedagogical intention
- executability
- first formalisation of advanced theory for many problems: matchings, flows, matroids, TSP
- uniform methodology and avoidance of redundancies
- sometimes re-formalisation of existing things
- major resources:
  - *Combinatorial Optimization* by Bernhard Korte and Jens Vygen
  - *Combinatorial optimization. Polyhedra and efficiency* by Alexander Schrijver
  - *LEDA: A Platform for Combinatorial and Geometric Computing* by Kurt Mehlhorn and Stefan Näher

- GitHub repo: `https://github.com/mabdula/Isabelle-Graph-Library`
- paper: *A Formal Analysis of Capacity Scaling Algorithms for Minimum Cost Flows*, ITP 2024
- by Mohammad Abdulaziz and myself

Other People's Formalisation Work (Selection)

- Dijkstra's SSP: Moore + Zhang (ACL2, 2005), Lee + Rudnicki (Mizar, 2005), Lammich + Nordhoff (2012, Isabelle) and Mohan et al. (Coq, 2021)
- Kruskal's for Minimum Spanning Trees: Haslbeck + Lammich + Biendarra (Isabelle, 2019)
- Maximum Flows: Lee (Mizar, 2005), Veltri (HOL Light, 2012) and Lammich + Sefidgar (Isabelle, 2016/17)
- Gale-Shapley for Stable Matching: Hamid + Castleberry (Coq, 2010) and Nipkow (Isabelle, 2021)

Table of Contents

Introduction and Aims

Network Flows

Mincost Flow Algorithms

Orlin's Algorithm

Formalisation Methodology

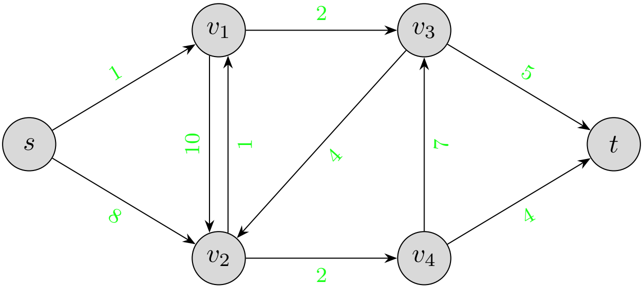
Running Time of Orlin's Algorithm

Limitations of Orlin's Algorithm

Summary

Network Flows: Maximum Flows

- a directed Graph (V, E).
- find $f : E \to \mathbb{R}_0^+$
- edge capacities u: $f(e) \leq u(e)$
- ingoing flow = outgoing flow, two designated vertices s and t
- send as much flow as possible from s to t.

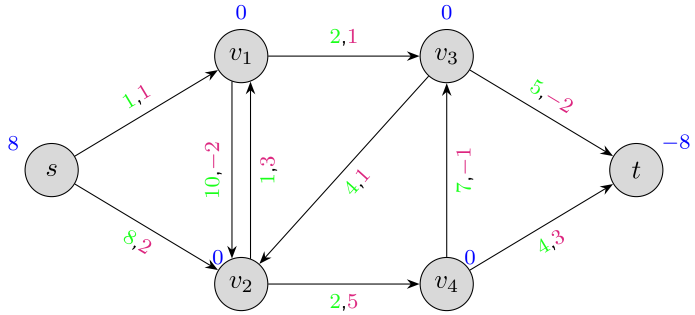


latex code taken from

`https://tex.stackexchange.com/questions/341949/how-to-plot-a-network-flow-with-tikz` **(modified)**

Network Flows: Minimum Cost Flows

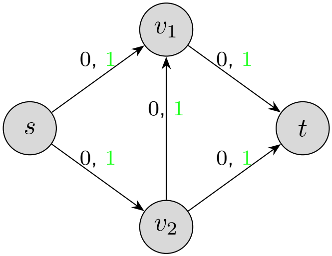
- a directed Graph (V, E).
- find $f : E \to \mathbb{R}_0^+$
- edge capacities u
- per-unit costs c for sending flow through an edge
- vertex balances b ($b(v) > 0$ 'supply', 'source'; $b(v) < 0$ 'demand', 'target')



- minimise $\sum_{e \in E} f(e) \cdot c(e)$
- typical application: sending goods around (fluids, electricity etc.)
- or: edge-disjoint paths, airline scheduling, baseball elimination, project selection
- computer vision: image smoothing

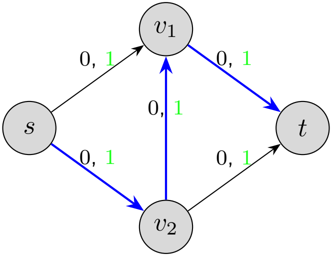
Network Flows: Finding an Optimum Solution

- improve solution iteratively
- greedy approach fails:



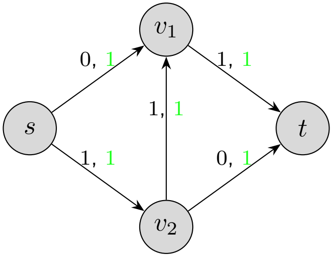
Network Flows: Finding an Optimum Solution

- improve solution iteratively
- greedy approach fails:



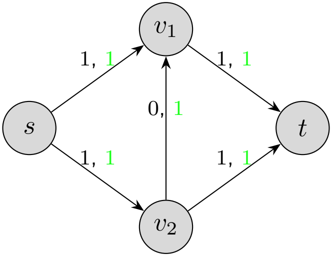
Network Flows: Finding an Optimum Solution

- improve solution iteratively
- greedy approach fails:



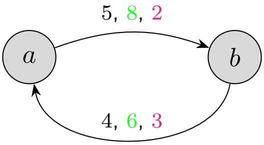
Network Flows: Finding an Optimum Solution

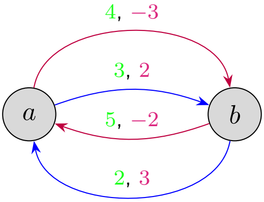
- improve solution iteratively
- greedy approach fails:



Network Flows: Residual Graphs

- strategy: improvement by adding and removing while maintaining feasibility
- augmentation as technique to send flow through the network
- for any original edge e, introduce forward edge e' and backward edge $\overleftarrow{e}$
- residual capacities $\mathfrak{u}$: $\mathfrak{u}(e') = u(e) - f(e)$ and $\mathfrak{u}(\overleftarrow{e}) = f(e)$
- residual costs $\mathfrak{c}$: $\mathfrak{c}(e') = c(e)$ and $\mathfrak{c}(\overleftarrow{e}) = -c(e)$





Augmentation for Flows

- augmentation = change flow assigned to original edges: $+\gamma$ for forward, $-\gamma$ for backward edges.
- augmenting path: path of residual edges with positive residual capacity ($\mathfrak{u}(P) = \min_{e \in P} \mathfrak{u}(e)$)
- augmentation along P
- characterisation: f is a maximum s-t-flow iff $\nexists$ augmenting path
- augmenting cycle: closed augmenting path with negative costs ($\mathfrak{c}(P) = \sum_{e \in P} \mathfrak{c}(e)$)
- effect of augmentation: $c(f') = c(f) + \gamma \cdot \mathfrak{c}(P)$
- characterisation: f is a mincost flow iff $\nexists$ augmenting cycle.
- formalised all these results

Table of Contents

Introduction and Aims

Network Flows

Mincost Flow Algorithms

Orlin's Algorithm

Formalisation Methodology

Running Time of Orlin's Algorithm

Limitations of Orlin's Algorithm

Summary

Mincost Flow Algorithms

- use augmentation to iteratively improve solution
- optimality follows from characterisation
- cycle cancelling: augment along mincost augmenting cycles
- shortest path: augment along minimum cost augmenting paths from sources to targets

A simple Algorithm

- take s with $b(s) > 0$, t with $b(t) < 0$, and
- a minimum cost augmenting path P connecting them
- augment P by $\gamma \in \mathbb{R}^+$ below residual capacity
- decrease supply/demand at s/t by γ
- bad running time

Correctness

- invariant[1]: capacity constraints satisfied: $0 \leq f(e) \leq u(e)$
- invariant: The current flow f does not allow for an augmenting cycle

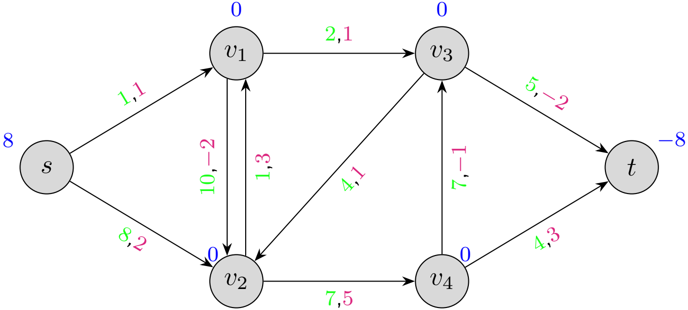
Theorem (KV 9.11)

If f does not contain an augmenting cycle and f is augmented along a minimum cost augmenting path P by γ, resulting in f', then f' does not have an augmenting cycle either.

- finally all flow distributed
- minimum cost flow obtained

[1] Invariant: A property always true at a certain line of a program. Way of induction over a loop execution.

let $\gamma = 1$



- 8 augmentations
- number of iterations linear in $\sum_{v \in V} |b(v)|$
- very inefficient

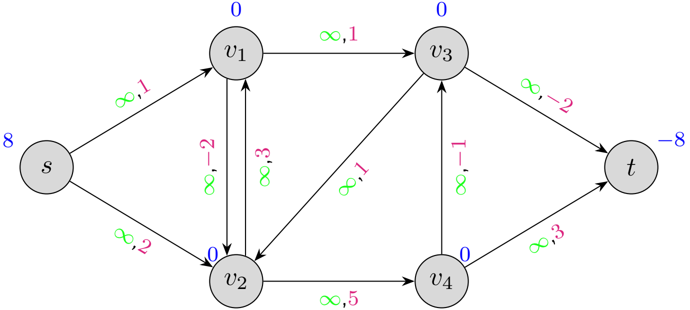
Capacity Scaling Algorithm

- infinite capacities + integer b
- sources + targets with high supply + demand
- fast progress
- sufficiently high balance: balance above threshold ($= \gamma$)

Initialise;
while *True* **do**
    while *True* **do**
        if $\forall v \in V.\ b(v) = 0$ **then return** current flow f;
        else if $\exists\, s\ t.\ b(s) > \gamma \wedge b(t) < -\gamma \wedge t$ *is reachable from* s **then**
            take such s, t, and a connecting minimum cost augmenting path P;
            augment f along P from s to t by γ;
            $b(s) \leftarrow b(s) - \gamma$; $b(t) \leftarrow b(t) + \gamma$;
        else if $\gamma = 1$ **then** no flow exists
        else break;
    $\gamma \leftarrow \frac{\gamma}{2}$;

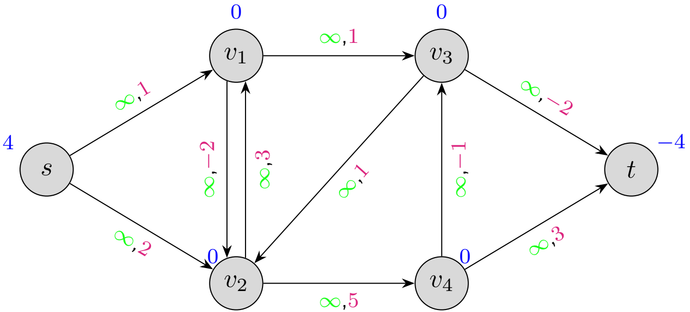
---

let $\gamma = 4$

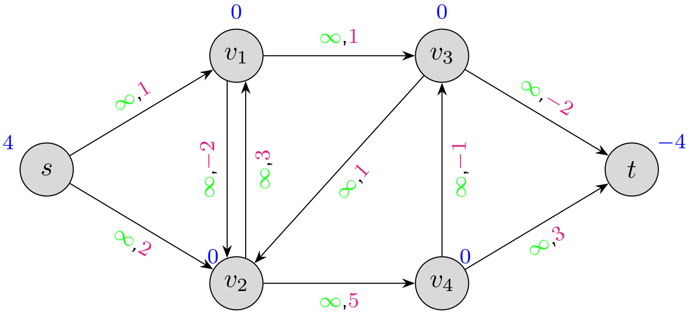


8
s
0
v_1
0
v_3
0
v_2
0
v_4
-8
t
∞,1
∞,2
∞,1
∞,-2
∞,3
∞,1
∞,-1
∞,5
∞,-2
∞,3

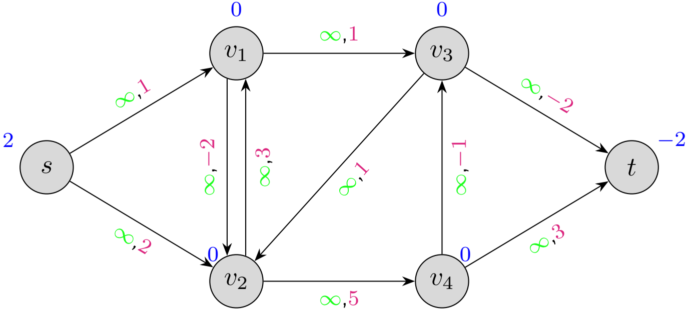
let $\gamma = 4$



let $\gamma = 2$

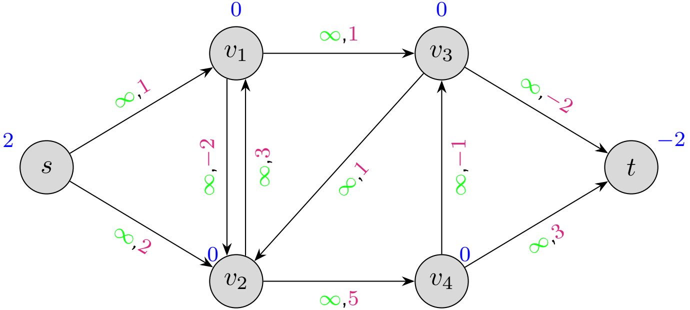


let $\gamma = 2$



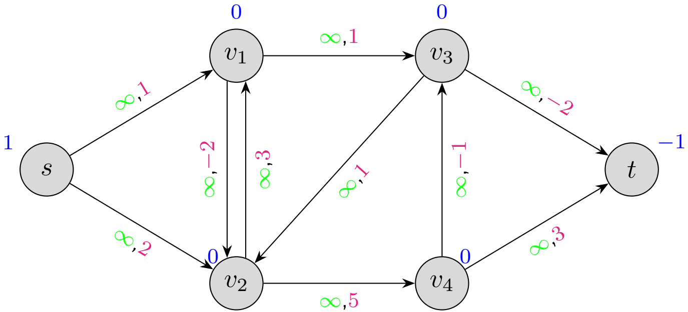
s
v_1
v_2
v_3
v_4
t
2
0
0
0
0
-2
∞,1
∞,2
∞,1
∞,-2
∞,3
∞,1
∞,5
∞,-1
∞,-2
∞,3

let $\gamma = 1$

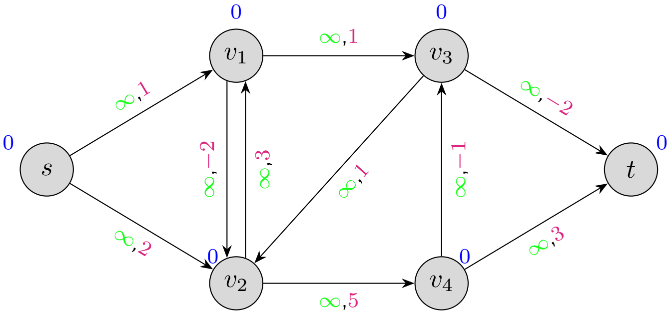


s
v_1
v_2
v_3
v_4
t
2
0
0
0
0
-2
∞,1
∞,2
∞,-2
∞,3
∞,1
∞,1
∞,5
∞,-1
∞,-2
∞,3

let $\gamma = 1$



let $\gamma = 1$



0
0
0
0
0
0
s
v_1
v_2
v_3
v_4
t
∞,1
∞,2
∞,−2
∞,3
∞,1
∞,1
∞,5
∞,−1
∞,−2
∞,3

- $\log 8 + 1$ augmentations
- number of iterations linear in $\log \sum_{v \in V} |b(v)|$
- much better

Table of Contents

Introduction and Aims

Network Flows

Mincost Flow Algorithms

Orlin's Algorithm

Formalisation Methodology

Running Time of Orlin's Algorithm

Limitations of Orlin's Algorithm

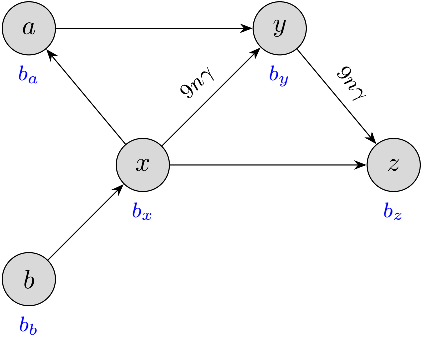
Summary

Orlin's Algorithm

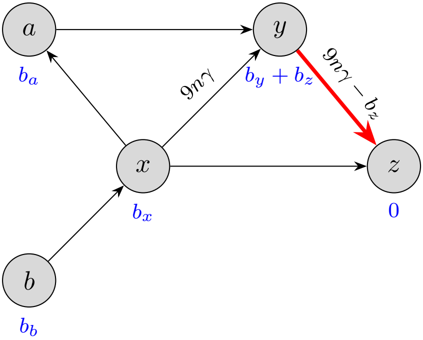
- concentrate balance at certain vertices (by augmentation)

Orlin's Algorithm: Concentrating Balances

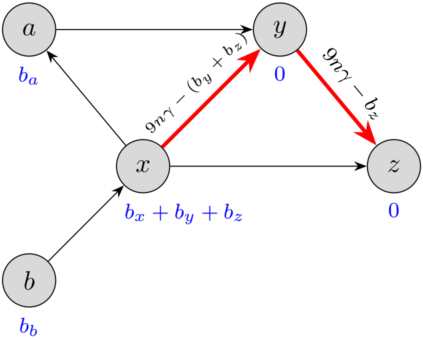
- building a graph of high-flow edges (forest)
- concentration: one non-zero vertex per component (representative)
- deactivate non-forest edges within components
- use of deactivated edges forbidden



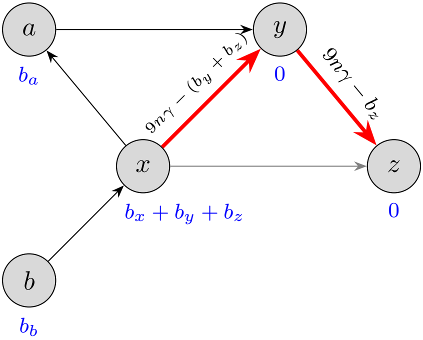
a
y
x
z
b
b_a
b_y
b_x
b_z
b_b
$g_{n\gamma}$
$g_{n\gamma}$



a
y
x
z
b
b_a
b_y + b_z
b_x
0
b_b
9nγ
9nγ − b_z



a
b_a
y
0
$9n\gamma - (b_y + b_z)$
$9n\gamma - b_z$
x
$b_x + b_y + b_z$
z
0
b
b_b



a
b_a
y
0
$9n\gamma - (b_y + b_z)$
$9n\gamma - b_z$
x
$b_x + b_y + b_z$
z
0
b
b_b

Why is that a good idea?

- sources and targets are representatives
- reduction by growing the forest
- time between component merges is *poly*(n, m)
- at most n merges
- number of augmentations is linear in *poly*(n, m)
- c, b, u irrelevant
- reduce number of times when we have to search for s, t and P
- considerable running time improvement
- (real balances allowed, capacities still infinite!)

Initialise;
while ***True*** **do**

- **while** ***True*** **do**
  - **if** $\forall v \in V.\ b'(v) = 0$ **then return current flow** f;
  - **else if** $\exists\, s.\ b'(s) > (1-\epsilon)\cdot\gamma$ **then**
    - **if** $\exists\, t\ .b'(t) < -\epsilon\cdot\gamma \wedge t$ ***is reachable from*** s **then**
      - **take such** s, t, **and a connecting minimum cost augmenting path** P **using active and forest edges only**;
      - **augment** f **along** P **from** s **to** t **by** γ;
      - $b'(s) \leftarrow b'(s) - \gamma$; $b'(t) \leftarrow b'(t) + \gamma$;
    - **else no suitable flow exists**;
  - **else if** $\exists\, t.\ b'(t) < -(1-\epsilon)\cdot\gamma$ **then**
    - **if** $\exists\, s\ .b'(s) > \epsilon\cdot\gamma \wedge t$ ***is reachable from*** s **then**
      - **take such** s, t, **and a connecting minimum cost augmenting path** P **using active and forest edges only**;
      - **augment** f **along** P **from** s **to** t **by** γ;
      - $b'(s) \leftarrow b'(s) - \gamma$; $b'(t) \leftarrow b'(t) + \gamma$;
    - **else no suitabe flow exists**;
  - **else break and return to top loop**;
- **if** $\forall$ ***still active*** $e.\ f(e) = 0$ **then**
  - $\gamma \leftarrow \min\{\frac{\gamma}{2}, \max_{v\in V} |b'(v)|\}$;
- **else**
  - $\gamma \leftarrow \frac{\gamma}{2}$;
- **while** $\exists$ ***active*** $e = (x, y)$ ***not in the forest*** $\mathcal{F}.\ f(e) > 8n\gamma$ **do**
  - $\mathcal{F} \leftarrow \mathcal{F} \cup \{e, \overleftarrow{e}\}$; **let** $x' = r(x)$ **and** $y' = r(y)$;
  - **wlog.** $|$**component of** $y| \geq |$**component of** $x|$;
  - **let** Q **be the path in** $\mathcal{F}$ **connecting** x' **and** y';
  - **if** $b'(x') > 0$ **then**
    - **augment** f **along** Q **by** $b'(x)$ **from** x' **to** y';
  - **else**
    - **augment** f **along** $\overleftarrow{Q}$ **by** $-b'(x)$ **from** y' **to** x';
  - $b'(y') \leftarrow b'(y') + b'(x')$; $b'(x') = 0$;
  - **foreach** $d = (u, v)$ ***still active and*** $\{r(u), r(v)\} = \{x', y'\}$ **do**
    - **deactivate** d;
  - **foreach** v ***reachable from*** y' ***in*** $\mathcal{F}$ **do**
    - **set** $r(v) = y'$;

Table of Contents

Introduction and Aims

Network Flows

Mincost Flow Algorithms

Orlin's Algorithm

Formalisation Methodology

Running Time of Orlin's Algorithm

Limitations of Orlin's Algorithm

Summary

Formalisation Methodology

- loops as recursive functions (Krauss' function package)
- program state = collection of variables, realised a records
- invariants: properties that are always satisfied at certain lines of the program
- induction for program verification
- locales to assume subprocedures
- abstract datatypes for executability

Methodology: Loops by Recursion

- a simpler example

```
function (domintros) DFS::('v, 'vset) DFS-state ⇒ ('v, 'vset)
    DFS-state where
  DFS dfs-state =
    (case (stack dfs-state) of (v # stack-tl) ⇒
      (if v = t then (dfs-state (|return := Reachable|))
       else ((if (𝒩_G v -_G (seen dfs-state)) ≠ ∅_N then
                let u = (sel ((𝒩_G v) -_G (seen dfs-state)));
                    stack' = u# (stack dfs-state);
                    seen' = insert u (seen dfs-state)
               in DFS (dfs-state (|stack := stack',
                                   seen := seen' |))
              else let stack' = stack-tl in
                   DFS (dfs-state (|stack := stack'|)))))
    | - ⇒ (dfs-state (|return := NotReachable|)))
```

Methodology: Locales for Subprocedures

```
locale Set =
fixes empty :: 's
 and insert :: 'a ⇒ 's ⇒ 's
 and  isin :: 's ⇒ 'a ⇒ bool
 and  set :: 's ⇒ 'a set
 and  invar :: 's ⇒ bool ...
assumes set-empty:   set empty = {}
 and set-isin:    invar s  ⟹  isin s x = (x ∈ set s)
 and set-insert:  invar s  ⟹  set(insert x s) = set s ∪ {x}
 and invar-empty: invar empty
 and invar-insert:invar s  ⟹  invar(insert x s) ...
```

Methodology: Locales for Subprocedures

```
locale Set =
fixes empty :: 's
 and insert :: 'a ⇒ 's ⇒ 's
 and  isin :: 's ⇒ 'a ⇒ bool
 and  set :: 's ⇒ 'a set
 and  invar :: 's ⇒ bool ...
assumes set-empty:   set empty = {}
 and set-isin:    invar s  ⟹  isin s x = (x ∈ set s)
 and set-insert:  invar s  ⟹  set(insert x s) = set s ∪ {x}
 and invar-empty: invar empty
 and invar-insert:invar s  ⟹  invar(insert x s) ...
```

Methodology: Summary and General Perspective

- stepwise refinement [Wirth 1971 + Hoare 1972]
- abstract datatypes [Wirth 1971, Hoare 1972, Liskov and Zilles 1974]
- locales for stepwise refinement [Nipkow 2015, Abdulaziz + Mehlhorn + Nipkow 2019, Maric 2020]

Table of Contents

Introduction and Aims

Network Flows

Mincost Flow Algorithms

Orlin's Algorithm

Formalisation Methodology

Running Time of Orlin's Algorithm

Limitations of Orlin's Algorithm

Summary

Running Time of Orlin's Algorithm

Orlin's is significant because:

- fastest method for minimum cost flows
- strongly polynomial, i.e. polynomial in $n + m$
- sophisticated, considerable part of the proofs in textbook by Korte and Vygen

Methodology to Formalise Time

- Isabelle functions are time-less
- define running time functions resembling the structure
- e.g. mergesort: $T(n) = 2 \cdot T(\frac{n}{2}) + c \cdot n$
- same approach for Orlin's
- assume times for loop bodies and subprocedures
- variation of an approach by Nipkow et al.

```
function (domintros) T-orlins ::
  nat ⇒ ('a, 'd, 'c, 'edge-type) Algo-state
    ⇒ nat  ×  ('a, 'd, 'c, 'edge-type) Algo-state where
(orlinsTime tt_OC  state) =
(if (return state = success) then (tt_OC, state)
else if (return state = failure) then (tt_OC, state)
else (let f = current-flow state; b = balance state;
          γ = current-γ state; E' = actives state;
          γ' = (if ∀ e ∈ to-set E'. f e = 0
                then min (γ / 2) (Max { |  b v |  . v ∈ 𝒱})
                else (γ / 2));
          state'time = loopAtime (state ⦇current-γ :=  γ' ⦈);
          state''time = loopBtime (prod.snd state'time)
      in
 ((t_OC + t_OB + prod.fst state'time + prod.fst state''time)
           +++ (T-orlins tt_OC (prod.snd state''time)))))
```

```
function (domintros) T-orlins ::
  nat ⇒ ('a, 'd, 'c, 'edge-type) Algo-state
    ⇒ nat  ×  ('a, 'd, 'c, 'edge-type) Algo-state where
(orlinsTime tt_OC  state) =
(if (return state = success) then (tt_OC, state)
else if (return state = failure) then (tt_OC, state)
else (let f = current-flow state; b = balance state;
          γ = current-γ state; E' = actives state;
          γ' = (if ∀ e ∈ to-set E'. f e = 0
                then min (γ / 2) (Max { |  b v |  . v ∈ 𝒱})
                else (γ / 2));
          state'time = loopAtime (state ⦇current-γ :=  γ' ⦈);
          state''time = loopBtime (prod.snd state'time)
      in
 ((t_OC + t_OB + prod.fst state'time + prod.fst state''time)
           +++ (T-orlins tt_OC (prod.snd state''time)))))
```

Formalising Running Time

$$\begin{array}{rrl} T_{orlins} \leq & & (n-1)\cdot(t_{Auf}+t_{AC}+t_{AB}+t_{BC}+t_{BB}+t_{Buf}) \\ & + & (n\cdot(\ell+k+2)-1)\cdot(t_{BF}+t_{BC}+t_{Buf} \\ & & \qquad\qquad\qquad\qquad +t_{Auf}+t_{AC}+t_{OC}+t_{OB}) \\ & + & ((\ell+1)\cdot(2\cdot n-1))\cdot(t_{BC}+t_{BB}+t_{Buf}) \\ & + & (t_{BF}+t_{BC}+t_{Buf})+t_{OC} \end{array}$$

($\ell = \lceil \log(4\cdot m\cdot n+(1-\epsilon)) - \log\epsilon \rceil + 1$ and $k = \lceil \log n \rceil + 3$,
usually $\epsilon = \frac{1}{n}$)

- semi-formal

Running Time: Asymptotics

- Orlin's for infinite capacities: $\mathcal{O}(n(\log n + \log m))$ augmentations
- each $\mathcal{O}(m)$ (unweighted), $\mathcal{O}(m + n \log n)$ (Dijkstra, weighted) or $\mathcal{O}(mn)$ (Bellman-Ford, weighted)
- resulting in $\mathcal{O}(n(\log n + \log m) \cdot (m + n \log n))$ or $\mathcal{O}(n(\log n + \log m) \cdot mn)$.

Table of Contents

Introduction and Aims

Network Flows

Mincost Flow Algorithms

Orlin's Algorithm

Formalisation Methodology

Running Time of Orlin's Algorithm

Limitations of Orlin's Algorithm

Summary

Limitations of Orlin's Algorithm

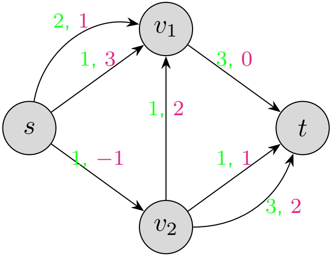
- central invariant: $\nexists$ augmenting cycle
- no negative cycles
- no capacities/infinite capacities only
- reduce other problems to that setting

Reduction

- $e = (x, y)$ with $u(e) < \infty$: add vertex e, edges (e, x) and (e, y), $b'(e) = u(e)$, $b'(x) = b(x) - u(e)$, $b'(y) = b(y)$
- for any flow in the old network, there is a flow in the new one and vice versa
- $\nexists$ negative cycle in new network iff $\nexists$ negative cycle in old network with infinite-capacity
- transform network
- compute minimum cost flow
- transform flow
- linear blowup
- Orlin's is fastest method for any flow problem

Flows in Multigraphs

- reduction requires multigraphs, formalisation changed



- set of objects with operations fst and snd
- algorithms not affected

- maxflow to mincost flow
- flow decomposition
- $\exists$ optimum flow iff feasible and $\nexists$ negative infinite-capacity cycle
- verified functional code for finite- and mixed-capacity minimum cost flows and maximum flows

Table of Contents

Introduction and Aims

Network Flows

Mincost Flow Algorithms

Orlin's Algorithm

Formalisation Methodology

Running Time of Orlin's Algorithm

Limitations of Orlin's Algorithm

Summary

Summary

- formalisation of Combinatorial Optimisation: Minimum Cost Flows, 35k LoP
- augmentation technique; also applicable to solve e.g. matching, matroid intersection
- characterisations of optimality
- scaling: pick sufficiently large parts first
- concentration/contraction and representatives
- Orlin's Algorithm formalised
- formalisation methodology: refinement
- semi-formal RT argument
- reductions among flow problems
- multigraphs

THANK YOU!