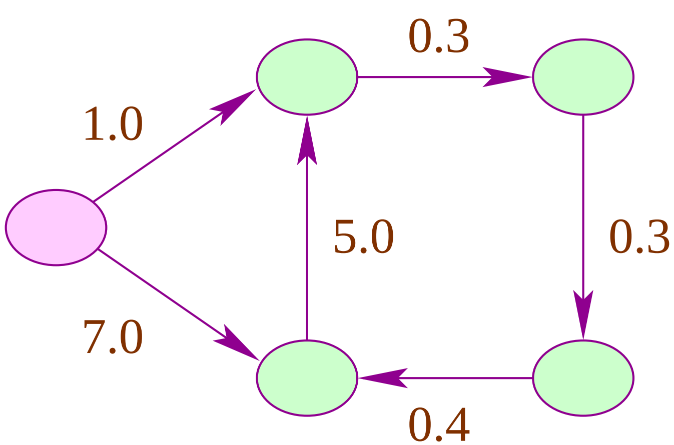
8.4 Kürzeste Wege

Gegeben:

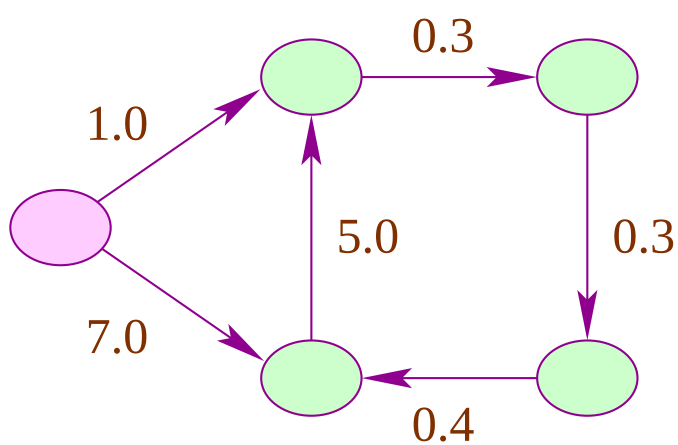
- Ein gerichteter Graph mit Kosten an den Kanten;
- ein Startknoten im Graphen.



8.4 Kürzeste Wege

Gegeben:

- Ein gerichteter Graph mit Kosten an den Kanten;
- ein Startknoten im Graphen.



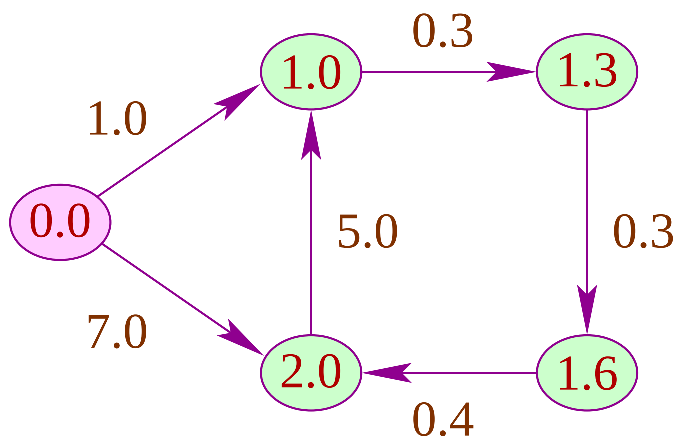
Gesucht:

Die minimalen Kosten, um die anderen Knoten zu erreichen ...

8.4 Kürzeste Wege

Gegeben:

- Ein gerichteter Graph mit Kosten an den Kanten;
- ein Startknoten im Graphen.



Gesucht:

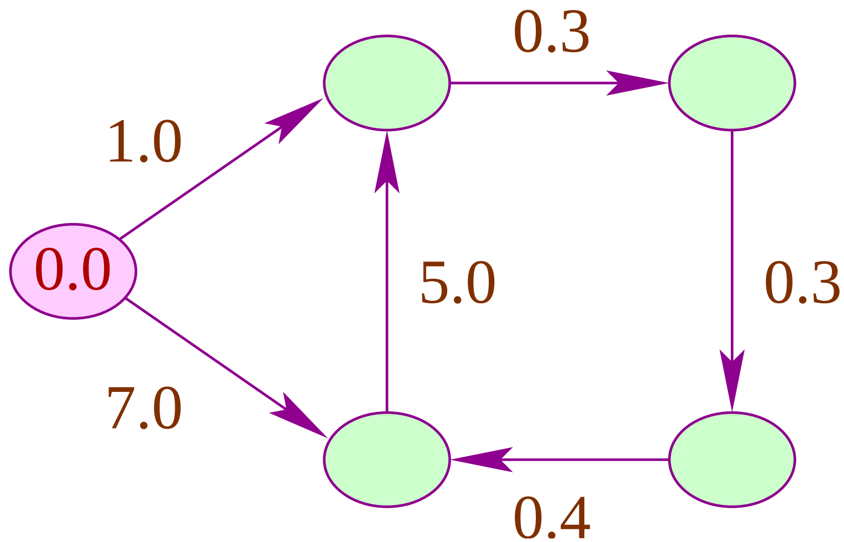
Die minimalen Kosten, um die anderen Knoten zu erreichen ...

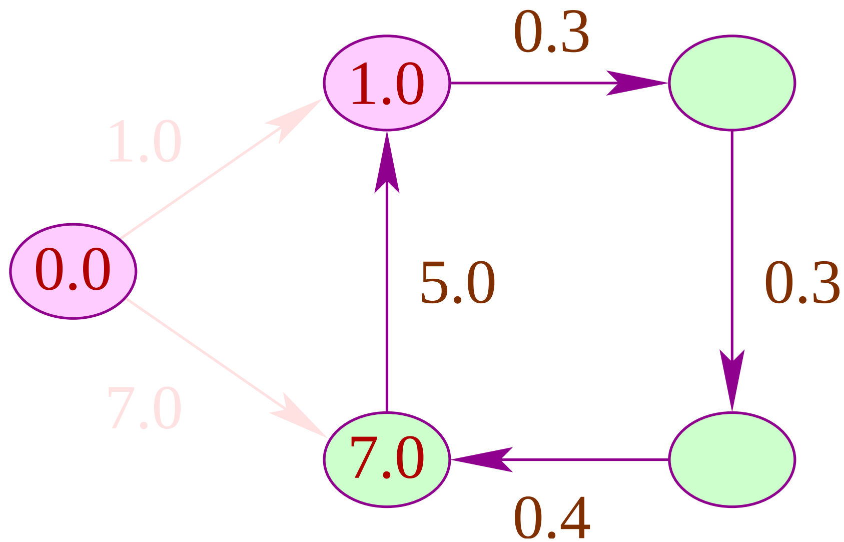
Idee: Dijkstras Algorithmus

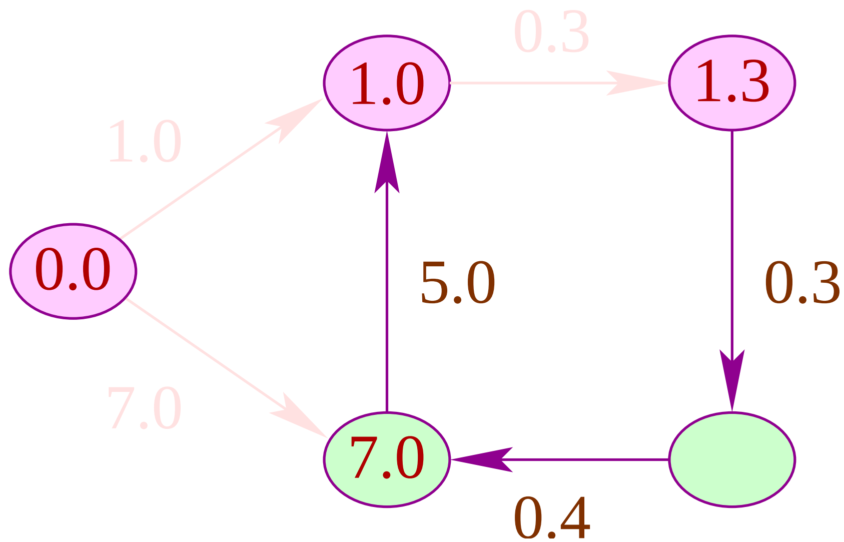
- Verwalte für jeden Knoten mögliche Kosten.
- Da die Kosten nicht-negativ sind, können die Kosten des billigsten Knotens können weiter reduziert werden !

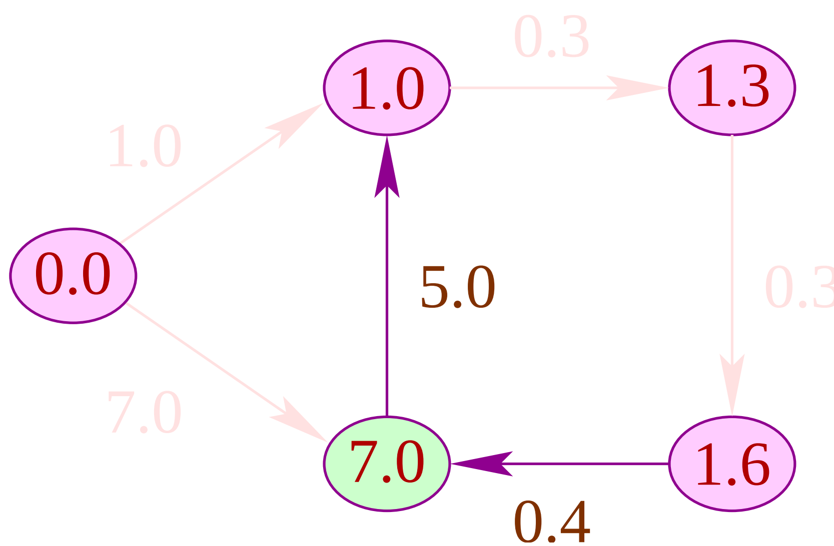
$\Longrightarrow$

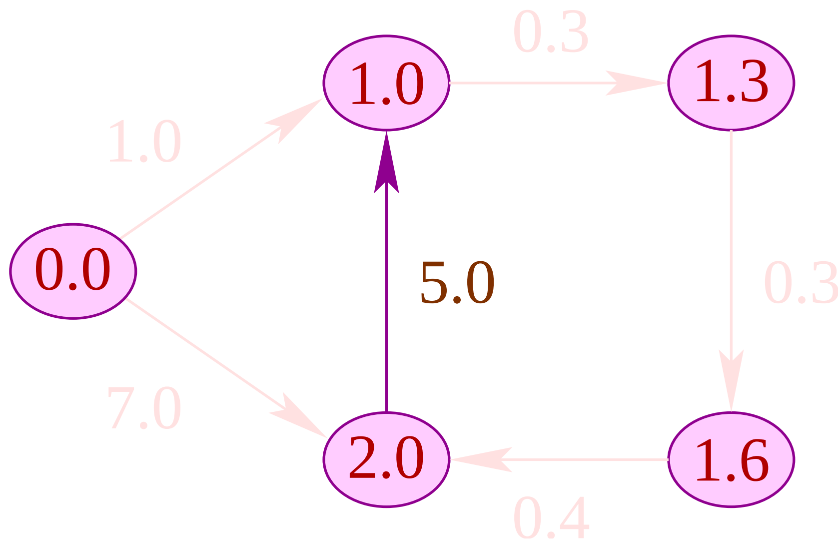
- Wir entfernen diesen Knoten;
- Wir benutzen seine ausgehenden Kanten, um die oberen Schranken der anderen Knoten evt. zu verbessern;
- Wir entfernen die Kanten und fahren mit dem nächst billigen Knoten fort ...

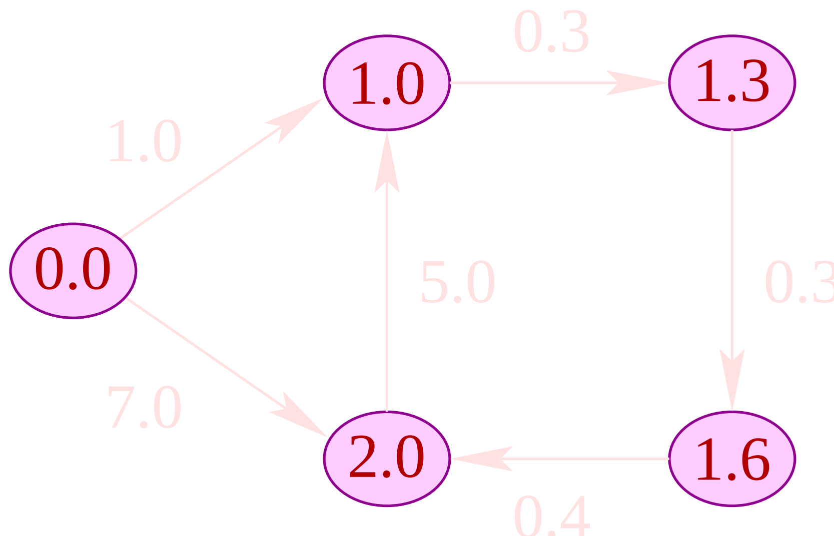

0.3
1.0
0.0
5.0
0.3
7.0
0.4


0.3
1.0
1.0
0.0
5.0
0.3
7.0
7.0
0.4


0.3
1.0
1.3
1.0
0.0
5.0
0.3
7.0
7.0
0.4


0.3
1.0
1.3
1.0
0.0
5.0
0.3
7.0
7.0
1.6
0.4


0.0
1.0
1.3
1.6
2.0
1.0
7.0
0.3
0.3
0.4
5.0


0.3
1.0
1.3
1.0
0.0
5.0
0.3
7.0
2.0
1.6
0.4

Edsgar Dijkstra, 1930-2002

Implementierung:

- Wir versuchen, stets die lokal günstigste Entscheidung zu treffen $\Longrightarrow$ greedy Algorithmus
- Wir verwalten die Menge der Paare `(c,x)` von bereits besuchten, aber noch nicht selektierten Knoten `x` mit aktuellen Kosten `c : float` in einer Datenstruktur.
- Die Datenstruktur `d` sollte die folgenden Operationen unterstützen:

```
extract_min : (float * node) d -> (float * node) option
insert : (float * node) -> (float * node) d -> (float * node) d
delete : (float * node) -> (float * node) d -> (float * node) d
```

$\Longrightarrow$ wir können AVL-Bäume benutzen :-)

```
# let dijkstra edges x = let n = Array.length edges
            in let dist = Array.make n None
            in let _ = dist.(x) <- Some 0.
            in let avl = Eq (Null, (0.,x), Null)
            in let do_edge (d,x) avl (a,y) = match dist.(y)
                   with None   -> dist.(y) <- Some (d+.a) ;
                                  insert (d+.a,y) avl
                   | Some old  -> if d+.a < old then (
                                     dist.(y) <- Some (d+.a);
                                     insert (d+.a,y) (
                                     delete (old,y) avl))
                                  else avl
            ...
```

```
...
in let rec bfs = function
       Null -> dist
     | avl  -> let (Some (d,x),avl) = extract_min avl
                in let avl  = List.fold_left
                                   (do_edge (d,x))
                                    avl
                                    edges.(x)
                in bfs avl
in bfs avl;;
```

Diskussion:

- Wurde `x` einmal als Minimum extrahiert, wird es nie wieder in `avl` eingetragen

  $\Longrightarrow$ es gibt maximal n `extract_min`

- Jede ausgehende Kante von `x` wird darum auch nur einmal behandelt :-))

  $\Longrightarrow$ es gibt maximal m `insert`

  $\Longrightarrow$ es gibt maximal m `delete`

- Der Gesamtaufwand ist folglich proportional zu:

$$n + m \cdot \log(n)$$

// das lässt sich mit Hilfe von Fibonacci-Heaps verbessern ;-)

9 Das Modulsystem von OCAML

$\longrightarrow$ Strukturen

$\longrightarrow$ Signaturen

$\longrightarrow$ Information Hiding

$\longrightarrow$ Funktoren

$\longrightarrow$ Constraints

$\longrightarrow$ getrennte Übersetzung

9.1 Module oder Strukturen

Zur Strukturierung großer Programmsysteme bietet Ocaml Strukturen an:

```
module Pairs =
  struct
    type 'a pair = 'a * 'a
    let pair (a,b) = (a,b)
    let first (a,b) = a
    let second (a,b) = b
  end
```

Auf diese Eingabe antwortet der Compiler mit dem Typ der Struktur, einer Signatur:

```
module Pairs :
  sig
    type 'a pair = 'a * 'a
    val pair : 'a * 'b -> 'a * 'b
    val first : 'a * 'b -> 'a
    val second : 'a * 'b -> 'b
  end
```

Die Definitionen innerhalb der Struktur sind außerhalb nicht sichtbar:

```
  # first;;
  Unbound value first
```

Zugriff auf Komponenten einer Struktur:

Über den Namen greift man auf die Komponenten einer Struktur zu:

```
# Pairs.first;;
- : 'a * 'b -> 'a = <fun>
```

So kann man z.B. mehrere Funktionen gleichen Namens definieren:

```
# module Triples = struct
    type 'a triple = Triple of 'a * 'a * 'a
    let first (Triple (a,_,_)) = a
    let second (Triple (_,b,_)) = b
    let third (Triple (_,_,c)) = c
  end;;
...
```

```
...
module Triples :
sig
  type 'a triple = Triple of 'a * 'a * 'a
  val first : 'a triple -> 'a
  val second : 'a triple -> 'a
  val third : 'a triple -> 'a
end
# Triples.first;;
- : 'a Triples.triple -> 'a = <fun>
```

... oder mehrere Implementierungen der gleichen Funktion:

```
# module Pairs2 =
  struct
    type 'a pair = bool -> 'a
    let pair (a,b) = fun x -> if x then a else b
    let first ab = ab true
    let second ab = ab false
  end;;
```

Öffnen von Strukturen

Um nicht immer den Strukturnamen verwenden zu müssen, kann man alle Definitionen einer Struktur auf einmal sichtbar machen:

```
# open Pairs2;;
# pair;;
- : 'a * 'a -> bool -> 'a = <fun>
# pair (4,3) true;;
- : int = 4
```

Sollen die Definitionen des anderen Moduls Bestandteil des gegenwärtigen Moduls sein, dann macht man sie mit `include` verfügbar ...

```
# module A = struct let x = 1 end;;
module A : sig val x : int end
# module B = struct
    open A
    let y = 2
  end;;
module B : sig val y : int end
# module C = struct
    include A
    include B
  end;;
module C : sig val x : int val y : int end
```

Geschachtelte Strukturen

Strukturen können selbst wieder Strukturen enthalten:

```
module Quads  = struct
    module Pairs = struct
        type 'a pair = 'a * 'a
        let pair (a,b) = (a,b)
        let first (a,_) = a
        let second (_,b) = b
      end
    type 'a quad = 'a Pairs.pair Pairs.pair
    let quad (a,b,c,d) =
        Pairs.pair (Pairs.pair (a,b), Pairs.pair (c,d))
    ...
```

```
    ...
    let first q = Pairs.first (Pairs.first q)
    let second q = Pairs.second (Pairs.first q)
    let third q = Pairs.first (Pairs.second q)
    let fourth q = Pairs.second (Pairs.second q)
  end

# Quads.quad (1,2,3,4);;
- : (int * int) * (int * int) = ((1,2),(3,4))
# Quads.Pairs.first;;
- : 'a * 'b -> 'a = <fun>
```

9.2 Modul-Typen oder Signaturen

Mithilfe von Signaturen kann man einschränken, was eine Struktur nach außen exportiert.

Beispiel:

```
module Count =
  struct
    let count = ref 0
    let setCounter n  = count := n
    let getCounter () = !count
    let incCounter () = count := !count+1
  end
```

Der Zähler ist nach außen sichtbar, zugreifbar und veränderbar:

```
# !Count.count;;
- : int = 0
# Count.count := 42;;
- : unit = ()
```

Will man, daß nur die Funktionen der Struktur auf ihn zugegreifen können, benutzt man einen Modul-Typ oder Signatur:

```
module type Count =
  sig
    val setCounter : int -> unit
    val incCounter : unit -> unit
    val getCounter : unit -> int
  end
```

Die Signatur enthält den Zähler selbst nicht :-)

Mit dieser Signatur wird die Schnittstelle der Struktur eingeschränkt:

```
# module SafeCount : Count = Count;;
module SafeCount : Count
# SafeCount.count;;
Unbound value Safecount.count
```

Die Signatur bestimmt, welche Definitionen exportiert werden. Das geht auch direkt bei der Definition:

```
module Count1 : Count =
  struct
    val count = ref 0
    fun setCounter n  = count := n
    fun getCounter () = !count
    fun incCounter () = count := !count+1
  end

# !Count1.count;;
Unbound value Count1.count
```

Signaturen und Typen

Die in der Signatur angegebenen Typen müssen Instanzen der für die exportierten Definitionen inferierten Typen sein :-)

Dadurch werden deren Typen spezialisiert:

```
module type A1 = sig
    val f : 'a -> 'b -> 'b
  end
module type A2 = sig
    val f : int -> char -> int
  end
module A = struct
    let f x y = x
  end
```

```
# module A1 : A1 = A;;
Signature mismatch:
Modules do not match: sig val f : 'a -> 'b -> 'a end
                                      is not included in A1
Values do not match:
  val f : 'a -> 'b -> 'a
is not included in
  val f : 'a -> 'b -> 'b
# module A2 : A2 = A;;
module A2 : A2
# A2.f;;
- : int -> char -> int = <fun>
```

9.3 Information Hiding

Aus Gründen der Modularität möchte man oft verhindern, dass die Struktur exportierter Typen einer Struktur von außen sichtbar ist.

Beispiel:

```
module ListQueue = struct
  type 'a queue = 'a list
  let empty_queue () = []
  let is_empty = function
      [] -> true | _  -> false
  let enqueue xs y = xs @ [y]
  let dequeue (x::xs) = (x,xs)
end
```

Mit einer Signatur kann man die Implementierung einer Queue verstecken:

```
module type Queue = sig
  type 'a queue
  val empty_queue : unit -> 'a queue
  val is_empty : 'a queue -> bool
  val enqueue : 'a queue -> 'a -> 'a queue
  val dequeue : 'a queue -> 'a * 'a queue
end
```

```
# module Queue : Queue = ListQueue;;
module Queue : Queue
# open Queue;;
# is_empty [];;
This expression has type 'a list but is here used with type
  'b queue = 'b Queue.queue
```

$\Longrightarrow$

Das Einschränken per Signatur genügt, um die wahre Natur des Typs `queue` zu verschleiern :-)

Soll der Datentyp mit seinen Konstruktoren dagegen exportiert werden, wiederholen wir seine Definition in der Signatur:

```
module type Queue =
sig
   type 'a queue = Queue of ('a list * 'a list)
   val empty_queue : unit -> 'a queue
   val is_empty : 'a queue -> bool
   val enqueue : 'a -> 'a queue -> 'a queue
   val dequeue : 'a queue -> 'a option * 'a queue
end
```

9.4 Funktoren

Da in Ocaml fast alles höherer Ordnung ist, wundert es nicht, dass es auch Strukturen höherer Ordung gibt: die Funktoren.

- Ein Funktor bekommt als Parameter eine Folge von Strukturen;
- der Rumpf eines Funktors ist eine Struktur, in der die Argumente des Funktors verwendet werden können;
- das Ergebnis ist eine neue Struktur, die abhängig von den Parametern definiert ist.