Informatik II
Van Emde Boas Trees

G. Zachmann
Clausthal University, Germany
zach@tu-clausthal.de

Einleitende Betrachtungen

- Bei vielen Algorithmen / Datenstrukturen (z.B. Sortieren) haben wir nur eine minimale Menge an Basisoperationen vorausgesetzt (z.B. die Vergleichsoperation) ...
- Aber: warum sollen wir nicht ausnutzen, dass wir in allen Fällen eine Maschinenrepräsentation der Daten haben und dass diese in den meisten Fällen
  - einfach ist (z.B. Binärzahlen), und
  - wir darauf noch viel mehr Operationen zur Verfügung haben? (in Zeit O(1), also Basisoperationen)

- Van-Emde-Boas-Trees haben alles, was ein Klassiker unter den Datenstrukturen braucht ☺ :
  - Ist hinreichend einfach, so dass eine relativ große Zahl von Leuten sie verstehen
  - Ist elegant
  - Hat Anwendungen in vielen sehr verschiedenen Bereichen
  - Hat einen hohen "*Impact*" (= wird in vielen anderen Algorithmen als Tool verwendet)
  - Löst ein fundamentales Problem optimal
  - Es gibt 101 Varianten davon
  - Und D.E. Knuth war begeistert davon ☺

- Portrait des Erfinders (von seiner Homepage http://staff.science.uva.nl/~peter)

Das Ziel

- Betrachte als Universum eine Teilmenge der ganzen Zahlen:

$$\mathcal{U} = \{0, \ldots, u-1\} \subset \mathbb{N}$$

- Gesucht: eine Datenstruktur T, die $S \subset \mathcal{U}$ speichern kann, und folgende Operationen für $x \in \mathcal{U}$ bietet:
  - T.find(x) : suche x in T
  - T.insert(x) : füge x zu T hinzu
  - T.delete(x) : lösche x aus T
  - T.findSucc(x) : suche das kleinste Element in T, das größer x ist, m.a.W., suche $\min\{y \in S \mid y > x\}$
  - T.findPred(x) : analog

Naive Ansätze

- Lösung mittels Bit-Array (a.k.a., Bit-String, Bit-Set)
- Definition:
  Ein Bit-Array a ∈ $\{0,1\}^u$ repräsentiert $S \subset \mathcal{U}$ folgendermaßen:

$$x \in S \Leftrightarrow a[x] = 1$$

- Beispiel: die Menge { 1, 9, 17 } aus dem Universum {0, 31} wird als Bit-String so repräsentiert

  01000000010000000100000000000000

  (LSB first).
- Vorteile: Insert und Delete gehen in Zeit $O(1)$
- Nachteile:
  - Für das Universum der Integer-Zahlen braucht man seehr lange Bitstrings
  - FindSucc geht (im *worst case*) nur in Zeit $O(u)$!

- Lösung mittels binärem Suchbaum:
  - Baue über dem (gedachten) Bitstring einen vollständigen BST auf
  - Die Blätter sind die Bits im Bitstring
  - Ein Knoten wird mit "1" beschriftet, falls eines der beiden Kinder eine "1" hat (logisches Oder)
- Beispiel:

```
1
1 1
1 1 1 0
1 0 1 0 1 0 0 0
1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

- Verbesserung: findSucc geht nun in $O(\log u)$

Es geht noch besser — mit vEB-Trees

Hilfsfunktionen

- Jedes $x \in \mathcal{U}$ wird als Binärzahl mit $\log(u)$ vielen Bits dargestellt
- Definiere:

$$\text{high}(x) = \lfloor \frac{x}{\sqrt{u}} \rfloor$$

$$\text{low}(x) = x \mod \sqrt{u}$$

- In Inforamtikersprache:

  high(x) := obere Hälfte der Bits der Binärdarstellung von x

  low(x) := untere Hälfte der Bits der Binärdarstellung von x

  - Man kann diese Operationen also sehr schnell implementieren

- Annahme im Folgenden: $u = 2^{2^k}$

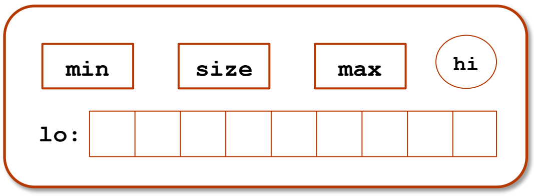
Die Datenstruktur

- Definition:
  Ein vEB-Tree B_u speichert eine Teilmenge $S \subset \mathcal{U}$ mit Hilfe folgender (interner) Daten:
  1. Array `lo` der Größe $\sqrt{u}$, wobei `lo[i]` ein vEB-Tree $B_{\sqrt{u}}$ ist;
  2. `hi` = ein vEB-Tree $B_{\sqrt{u}}$
  3. `size` = Anzahl Elemente in B
  4. `max`, `min` = größtes und kleines Element aus S

- Achtung: $B_{\sqrt{u}}$ ist "nur" noch ein vEB-Tree über ein Universum

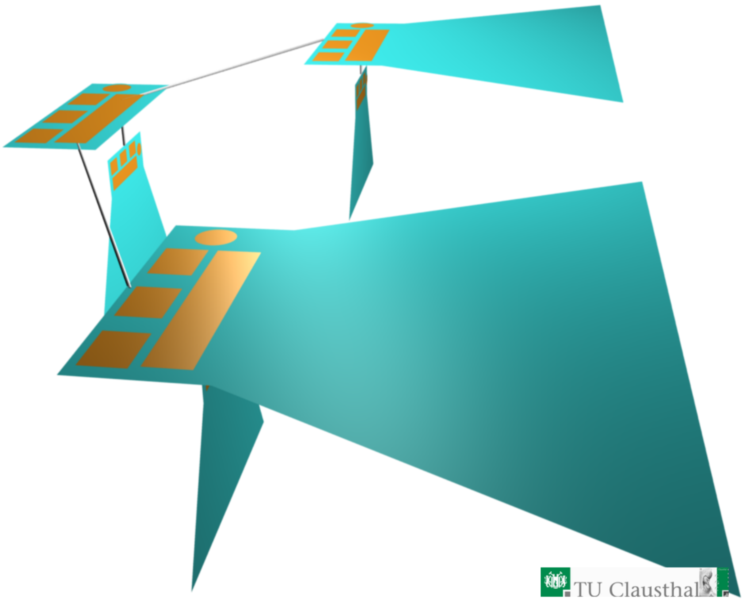
$$\mathcal{U}' = \{0, \ldots, \sqrt{u} - 1\}$$

- Ein Knoten eines vEB-Tree sieht also so aus:



- Repräsentation der Menge *S* im vEB-Tree *B*:
  - Min und Max von S werden in **`min, max`** gespeichert
  - Alle übrigen $x \in S \setminus \{\text{min}, \text{max}\}$ werden so gespeichert:
    - Zerlege $x =: a\sqrt{u} + b$
    - **`hi`** speichert den Wert *a*, **`lo[a]`** speichert den Wert *b*
- Bemerkung: man hat also den zusätzlichen Constraint, daß der vEB-Tree **`hi`** den Wert *a* genau dann enthält, wenn der vEB-Tree **`lo[a]`** nicht leer ist.
- Noch eine Bemerkung:
  - Der vEB-Tree **`lo[a]`** ist also für alle Werte im Bereich $a\sqrt{u}, \ldots, (a+1)\sqrt{u} - 1$ "zuständig"
  - Bei allen diesen Werten sind die oberen $\frac{\log u}{2}$ Bits gleich

Graphical Representation of a vEB-Tree



Die Insert-Operation

- Der Algorithmus:

```
insert( x ):
  if size == 0:
    size = 1
    min = max = x
  else
    if x < min:
      min, x = x, min              # (1)
    a = high(x); b = low(x)
    if lo[a] == None:
      hi.insert( a )
      lo[a] = vEB_Tree()
    lo[a].insert( b )              # (2)
    update size & max
```

- Bemerkungen:
  1) Hier wird **x** in **min** gespeichert (als neues **min**), so daß man im Folgenden nur noch das alte **min** woanders unterbringen muß
  2) Falls **lo[a]** vorher leer war, kostet dieser Aufruf nur Zeit $O(1)$!

Die FindSuccessor-Operation

- Der Algorithmus:

```
findSucc( x ):
  test and deal with special cases
    (size==0, size==1, size==2)
  if x < min:
    return min
  a = high(x); b = low(x)
  if lo[a] != None and b < lo[a].max:
    c = a
    d = lo[a].findSucc( b )          (1)
  else
    c = hi.findSucc( a )             (2)
    d = lo[c].min
  y = c*sqrt(u) + d
  return y
```

- Bemerkungen:
  1) Liefert auf jeden Fall einen gültigen Wert `d`
  2) Aus Gründen der Klarheit fehlt hier der Fall, dass in `hi` kein Nachfolger zu `a` existiert

Ein vEB-Tree ist gleichzeitig eine P-Queue

- Erinnerung: P-Queue = Datenstruktur, die folgende Operationen erlaubt
  - Insert(item, prio)
  - ExtractMin → liefert dasjenige Element mit höchster Prio
- Delete auf vEB-Trees funktioniert analog zur Insert-Operation
- Damit kann man trivial eine ExtractMin-Operation implementieren
- Damit realisiert ein vEB-Tree eine P-Queue und erlaubt
  - Insert in Zeit $O(\log \log n)$
  - ExtractMin in Zeit $O(\log \log n)$
  - Ist also exponentiell besser als Heaps

Laufzeit

- Rekurrenzgleichung für die Laufzeit lautet:

$$T(n) \le T(u) = T(\sqrt{u}) + O(1)$$

- Auflösen:

$$\begin{aligned} T(u) &= T(u^{\frac{1}{2}}) + O(1) \\ &= T(u^{\frac{1}{4}}) + 2 \cdot O(1) \\ &= T(u^{\frac{1}{8}}) + 3 \cdot O(1) \\ &= T(u^{\frac{1}{2^k}}) + k \cdot O(1) \end{aligned}$$

- Ergibt:

$$T(n) \in O(\log \log u)$$

Platzbedarf

- Rekurrenzgleichung (für worst-case):

$$S(u) = \sqrt{u} \cdot \underbrace{S(\sqrt{u})}_{\text{Platz für ein lo[i]}} + \underbrace{S(\sqrt{u})}_{\text{Platz für hi}} + \underbrace{O(\sqrt{u})}_{\text{Platz für das Array lo}} + \underbrace{O(1)}_{\text{size, min, max}}$$

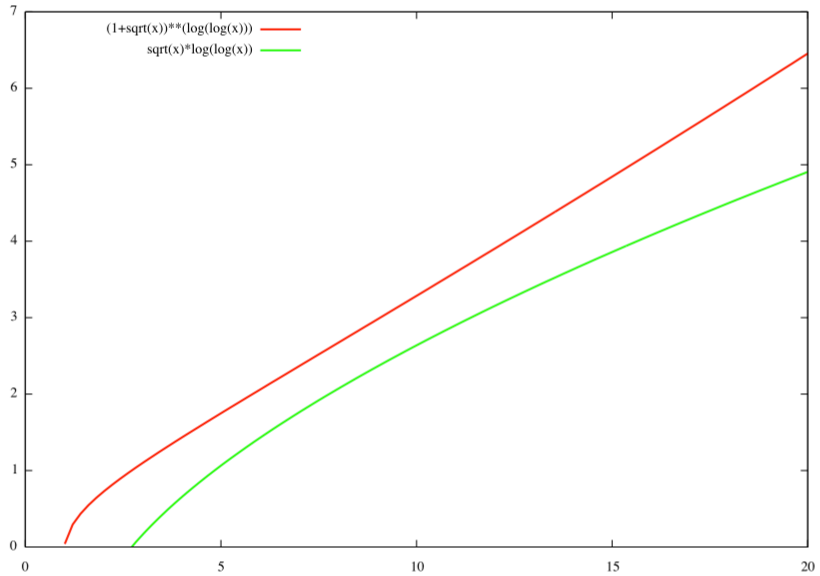
- Auflösen:

$$S(u) \in (1 + \sqrt{u}) S(\sqrt{u}) + O(\sqrt{u})$$

$$S(u) \in (1 + \sqrt{u})^k S(u^{\frac{1}{2^k}}) + k \cdot O(\sqrt{u})$$

$$S(u) \in (1 + \sqrt{u})^{\log \log u} + \log \log(u) \cdot O(\sqrt{u})$$

Ein Plot der Funktionen



Beweis, dass $S(u)$ linear ist

- Zunächst die verborgenen Konstanten "angenehm" machen:

$$S(u) = c_1(1+\sqrt{u})S(\sqrt{u}) + c_2\sqrt{u}$$

setze $$S'(u) = \frac{1}{\max(c_1, c_2)} S(u)$$

Damit wird $$S'(u) \leq (1+\sqrt{u})S'(\sqrt{u}) + \sqrt{u}$$

- Ansatz:

$$S'(u) \leq u + b$$

$$S'(4) = 1$$

- Beweis per Induktion:
  - Ind.anfang:

$$S'(4) = 1 \leq 4 + b$$

→ ja, falls $b \geq -3$

  - Ind.schritt:

$$\begin{aligned} S'(u) &\leq (1+\sqrt{u})S'(\sqrt{u}) + \sqrt{u} \\ &\leq (1+\sqrt{u})(\sqrt{u}+b) + \sqrt{u} \\ &\leq \sqrt{u}(2+b) + u + b \\ &\overset{!}{\leq} u + b \end{aligned}$$

→ ja, falls $b \leq -2$

- Zusammen: $S'(u) \leq u - 2$

Bemerkungen

- In der Praxis implementiert man vEB-Trees als Mix:
  - Baue vEB-Trees rekursiv auf
  - Sobald |S| < Wortbreite (32 / 64 Bit): schalte auf Bit-Array um
    - Viele CPUs bieten als Maschinenbefehle *shift-by-k* und *find-first-zero* mit wenigen Taktzyklen Ausführungszeit
- Selbstverständlich macht man *lazy initialization*:
  - Ein Kind-vEB-Tree wird nur erzeugt, wenn er wirklich Elemente enthält!
  - M.a.W.: ein leerer vEB-Tree besteht nur aus einem (Wurzel-)Knoten, dessen `lo`-Array lauter NULL-Pointer enthält (dito natürlich für den `hi`-Pointer)
  - (In vielen Darstellungen klingt es anders, ist aber Quatsch)

Eine Variante des vEB-Trees

- Situation: oft ist $n \ll \sqrt{u}$
- Annahmen:
  - $n = u^{\frac{1}{2^k}}$
  - Alle Keys sind völlig gleichmäßig über das Universum $\mathcal{U}$ verteilt
- Der vEB-Tree über der Menge S sieht nun grob so aus:
  - Das `hi`-Array enthält n Keys
  - Pro "√u-Fach" hat man höchstens einen Key (im Mittel)
  - Ein Element des `lo`-Arrays enthält einen NULL-Zeiger, falls in dem zugehörigen vEB-Tree kein Element aus S liegen würde

- Versucht man damit, die Space-Complexity von "einfachen" vEB-Trees abzuschätzen, kommt folgendes heraus:

$$S(n,u) = \sqrt{u} + S(1,\sqrt{u})\cdot n + S(n,\sqrt{u})$$

$$= S(n,\sqrt{u}) + \sqrt{u} + n$$

$$\vdots$$

$$= S(n, u^{\frac{1}{2^k}}) + kn + \sum_{i=1}^{k} u^{\frac{1}{2^i}}$$

$$\in O(n) + k\cdot n + O(u)$$

Eine Variante des vEB-Trees

- Ersetze das `lo`-Array durch eine Hash-Tabelle mit m Slots, so daß

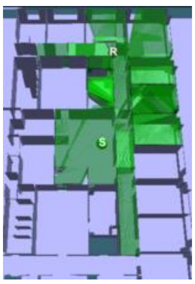
$$\frac{n}{m} = \alpha$$

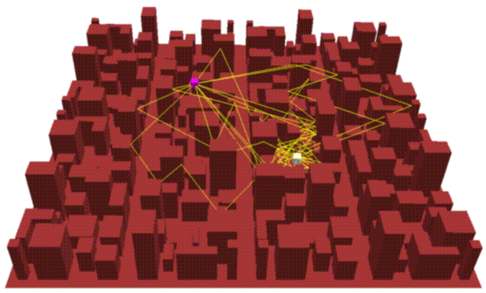
- Damit wird

$$\begin{aligned} S(n,u) &= \frac{n}{\alpha} + S(1,\sqrt{u})\cdot n + S(n,\sqrt{u}) \\ &\leq c\cdot n + S(n,\sqrt{u}) \\ &\leq k\cdot cn + S(n, u^{\frac{1}{2^k}}) \\ &\in O\big(\log\log(u)\cdot n\big) \end{aligned}$$

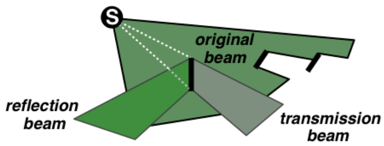
Anwendung: Akustische Modellierung

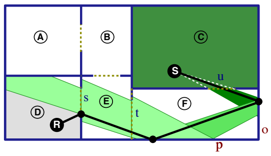
- Gegeben:
  - Ein Raum (oder viele) als geometrisches Modell (Menge von Polygonen)
  - Eine Schallquelle (*Source*)
  - Ein Empfänger (*Listener / Receiver*)
- Aufgabe: berechne den Schall, der beim Receiver ankommt
- Problem: der Schall durchläuft sehr viele Wege!





- Modellierung der Pfade des Schalls mittels Beams:

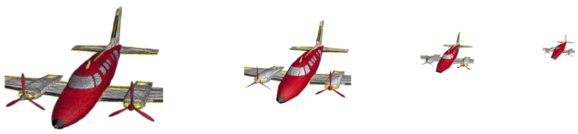




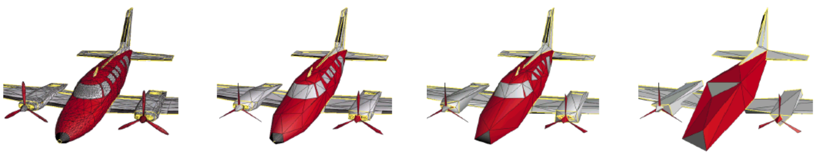
- *Alle* Pfade verfolgen dauert zu lange
- Idee: verfolge nur die Pfade, die für das Ohr wichtig sind
  - Kurze Pfade → kommen zuerst am Ohr an; geben wichtige Information über die räumliche Umgebung
  - Pfade mit starkem Schall → übertragen Infos über die Schallquelle, maskieren schwächere Signale
- Lösung: P-Queue von Beams, priorisiert nach Lautstärke und potentieller Länge

Levels of Detail in der Computer-Graphik

- Fällt Ihnen etwas auf?

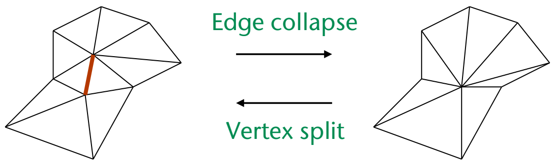


- Hoffentlich nicht ☺
- In Wahrheit haben die Modelle sehr verschiedene Auflösungen:



- Diese heißen *Level of Detail (LOD)*

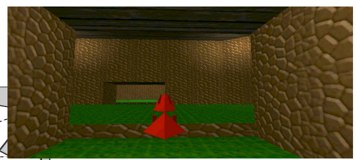
- Die fundamentale Operation bei der Simplifizierung:



- Führe Bewertungsfunktion für Kanten ein → Priorität
  - Wie wichtig ist die Kante für das Erscheinungsbild?
    - Wahrnehmungsbasiert (Krümmung, Viewpoint, Textur, ...)
- Verwalte alle noch verbleibenden Kanten des Modells in einer P-Queue; kollabiere als nächstes immer diejenige mit max. Priorität
- Achtung: die P-Queue muß hier zusätzlich unterstützen, daß die Priority-Keys der Einträge sich *nachträglich* ändern!

Pathfinding

- Aufgabe: autonome Figuren auf ein Ziel zubewegen



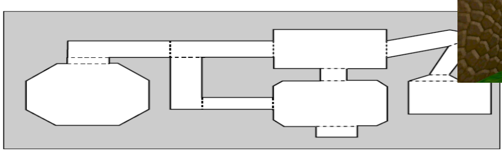


Figure 12.7 Dotted lines represent the portals between convex areas.

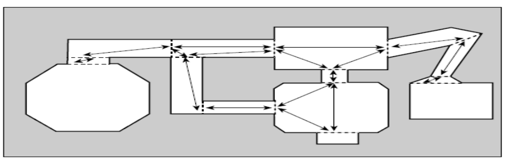
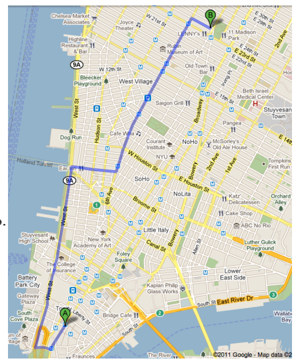
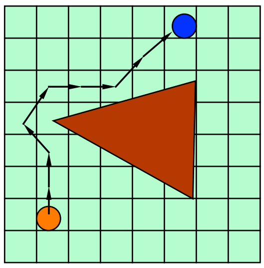


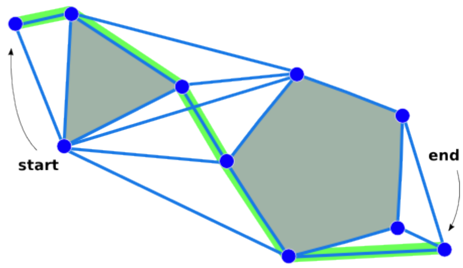
Figure 12.8 The portals can be the nodes in a graph representing the map.



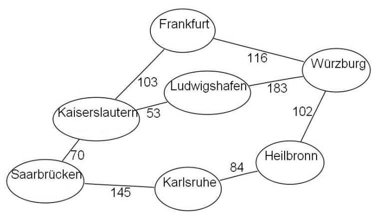
- Oder: Routenplanung im Straßennetz

- Repräsentation des Geländes mit Hindernissen:
  - Als Graph mit Knoten und Kanten
  - Oder als Gitter mit "verbotenen" Zellen; ist auch ein Graph:
    - Gitterzellen sind die Knoten
    - Jede Zelle ist über 4 Kanten mit den Nachbarn verbunden (falls kein Hindernis)
- Im Folgenden: betrachte nur noch Graphen





- Gegeben:
  - Ein Graph mit einer Menge $\mathcal{V}$ von Knoten, einer Menge Kanten, und Länge pro Kante
  - Ein Startknoten *S*
  - Ein Zielknoten *E* (*goal*)
- Gesucht: ein möglichst kurzer Pfad von *S* nach *E*
- Mögliche Lösungen:
  - Exhaustive Search mittels Backtracking bzw. Depth-First Search (zu teuer)
  - Dijkstra's Algorithmus (macht stumpfsinnig breadth-first search)
  - Der A*-Algorithmus



Der A*-Algorithmus

- Verwende zusätzliche Informationen (*informed search*)
- Annahme hier:
  - Jeder Knoten *V* hat eine Position (V_x, V_y) in der Ebene
  - Es gibt eine Heuristic *h*

$$h : \mathcal{V} \to \mathbb{R}$$

  die zu jedem Knoten *V* eine untere Schranke für den kürzesten Pfad von *V* nach *E* liefert
  - Zum Beispiel:

$$h(V) = \|V - E\|$$

- Klar ist: es gibt eine Funktion *g*, so daß
  g(*V*) = Länge des kürzesten Pfades von *S* nach *V*
  - Diese konstruieren wir im Folgenden peu à peu

- Definiere damit die "heuristisch geschätzten" Kosten eines Knotens *V* als

$$f(v) = g(v) + h(v)$$

- Die Idee des A*-Algorithmus:
  - Ausgangssituation:
    Menge $\mathcal{C} \subset \mathcal{V}$: alle Knoten, für die g(*V*) schon bestimmt wurde, und deren nächste Nachfolger auf dem Weg zu *E* schon festgelegt wurden;
    Menge $\mathcal{O} \subset \mathcal{V}$: alle Knoten, für die g(*V*) schon geschätzt wurde, und für die die Nachfolger noch *nicht* bestimmt wurden
  - Die Iteration (ganz grob):
    - Wähle aus *O* den Knoten V mit kleinstem Wert *f*(*V*) (P-Queue!)
    - Füge *V* zu *C* hinzu
    - Bestimme die Nachfolger von *V* → füge diese zu *O* hinzu

- A* ist somit eine Art "gerichteter Breadth-First Search"
- Terminologie:
  - C heißt "*closed set*"
  - O heißt "*open set*" – stellt quasi den "Rand" des bislang explorierten Terrains dar

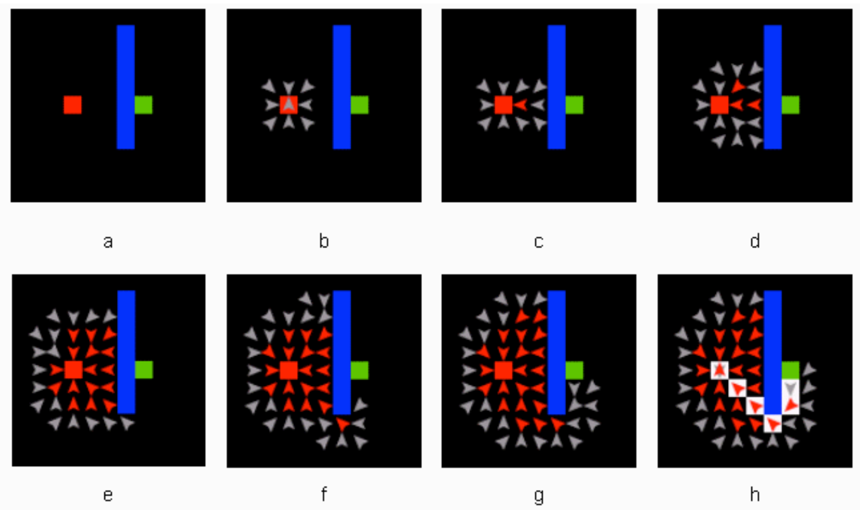
A* in Pseduo-Code

```
C := {}                          # closed set (as bit-set)
O := {S}                         # open set (as p-queue)
g(S) := 0; f(S) := h(S)
while O is not empty:
  V := extractMin( O )
  if V == E:
    goal reached; reconstruct path; return
  C.add( V )                     # won't find better S→V
  for all neighbors W of V:
    if W ∈ C:
      continue with next W
    if W ∉ O:
      g(W) := g(V) + d(V,W)      # d(V,W) = edge length
      f(W) := g(W) + h(W)        # first estimate of f(W)
      O.add( W, f(W) )           # add W to "fringe"
    else:                        # path S→W already known
      if g(V) + d(V,W) < g(W):
        g(W) := g(V) + d(V,W)    # better path found
        f(W) := g(W) + h(W)      # update f and p-queue!
      else:
        # path to W via V is no better, do nothing
return "no path from S to E exists"
```

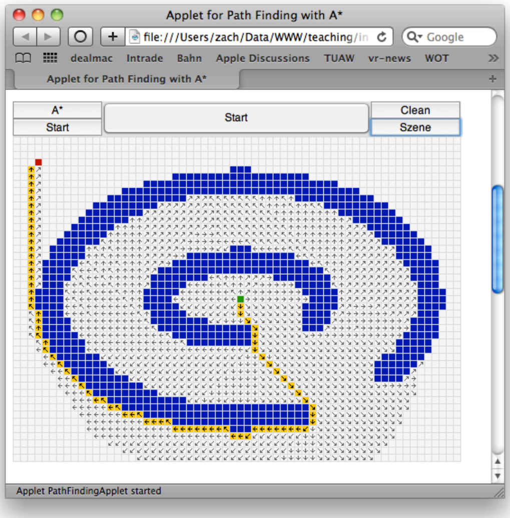
Informeller Beweise

- Zur Terminierung:
  Anzahl Knoten im Graphen ist endlich, jeder Knoten wird nur 1x in die P-Queue (O) aufgenommen
- Zur Korrektheit:
  - Falls es einen Pfad S → E gibt (oder mehrere), wird einer davon auch tatsächlich von A* gefunden
  - Beweis: jeder Knoten wird 1x besucht; einer davon ist E
- Zur Optimalität:
  - Wenn A* an einem Knoten V steht, dann haben alle Pfade $S \rightarrow V \rightarrow E$ mindestens die Kosten $f(V)$, wegen der Bedingung an h
  - V war aber von allen Knoten an der "Front" derjenige, der den kürzesten Pfad $S \rightarrow V$ hat, wegen der P-Queue

Ein Beispiel



Demo


Applet for Path Finding with A*
file:///Users/zach/Data/WWW/teaching/
Google
dealmac
Intrade
Bahn
Apple Discussions
TUAW
vr-news
WOT
Applet for Path Finding with A*
A*
Start
Start
Clean
Szene
Applet PathFindingApplet started