6. Kapitel

Speicherbereinigung

Kapitel 6: Speicherbereinigung

6.1 Speicherbereinigung (*SB*)

Ziel:

Wiederverwendung des Platzes nicht mehr benötigter Haldenobjekte

Speicherloch: Überflüssige Objekte bleiben erhalten

Probleme:

Aufwand (*evtl.*) beträchtlich (mehr als 10% Rechenzeit)

Bei Echtzeit-und interaktiven Aufgaben Betriebsunterbrechung vermeiden

Voraussetzung:

Alle Verweise auf überflüssige Objekte gelöscht

(Aufgabe des Anwenders, nicht trivial !)

6.1 SB: Alternativen

SB von Hand:

- explizite Speicherverwaltung, SB durch Anweisungen im Programm

automatische SB ohne Info:

- Konservative SB, (C, C++,...): Bitmuster prüfen, ob sie Verweise sein könnten (Boehm's SB)
- Referenzzähler: Jedes Objekt enthält Zähler der Verweise und seinen Umfang

automatische SB mit Info vom Übersetzer

- Urform: LISP-SB, Position sämtlicher Verweise und Objektumfang vorab bekannt
- die hier behandelten Verfahren

6.1 SB von Hand

- *mark-dispose* (Pascal,...): markiere Haldenpegel, beseitige später alle Objekte jenseits der Marke
  - Keller von Marken erforderlich
  - nur wenn *alle* Objekte zwischen Markieren und Beseitigen überflüssig
- Explizite Freilistenverwaltung mit expliziten Freigabeanweisungen im Programm
- Beide Verfahren
  - erfordern große Aufmerksamkeit
  - sind sehr fehleranfällig und wartungsunfreundlich
  - sind bereits für mittelgroße Programme ungeeignet

6.1 automatische SB ohne Info

Boehms SB, (C, C++,...):

- Bitmuster prüfen, ob sie Verweise sein könnten:
  "Wenn es ein Zeiger sein könnte, behandle es als Zeiger!"
- entscheide konservativ, welche (Register-/Keller-/Halden-)Inhalte Referenzen sein könnten
- bestimme Länge der Bezugsobjekte, beseitige alle anderen Objekte
- Verschiebung von Objekten nicht möglich
  - Externe Fragmentierung
- "Speicherlöcher" möglich
  - Gleitkommazahlen, Text sieht oft wie gültige Speicheradresse aus
  - praktisch immer begrenzt (typ. 10%)

6.1 automatische SB ohne Info

Referenzzähler:

- Jedes Objekt enthält Zähler der Verweise und seinen Umfang
- bei Zuweisungen an Verweisvariable: Zähler des bisherigen Bezugsobjekts herunter-, des neuen Bezugsobjekts heraufzählen
- Objekt beseitigen, wenn Zähler null , d.h. seinen Platz in eine Freiliste aufnehmen
  - Hoher Zusatzaufwand bei jeder Zuweisung von Verweisen
  - keine Verschiebung von Objekten möglich
  - Umfang der Referenzzähler?
  - Überlauf der Referenzzähler?
  - Speicherlöcher: Zyklische Listen werden nicht beseitigt
- Faule Referenzzähler: Übersetzer erkennt und eliminiert Aktualisierungen, die sich gegenseitig aufheben

```
tmp = p;
p   = q;
q   = tmp;
```

6.1 automatische SB mit Info

finde *alle* überflüssigen Objekte und beseitige sie mit Hilfe von

- Freilistenverwaltung,
- Zusammenschieben oder
- Kopieren der benötigten Objekte

Voraussetzung: Lage aller Verweise und Umfang aller Objekte bekannt

- entweder aus Systementwurf, z.B. alle Objekte gleich aufgebaut, gleich lang, Verweise an festen Stellen
- durch Selbstidentifikation:
  - alle Objekte (auch im Keller) enthalten Typkennung oder Länge und Positionsangaben der enthaltenen Verweise
  - bei polymorphen Objekten immer möglich
- durch Färbung der Verweise: alle Verweise sind (explizit oder implizit) mit dem Typ des Bezugsobjekts gekennzeichnet
  - explizit: Verweis umfaßt Typkennung des Bezugsobjekts
  - implizit: Typbeschreibung enthält Typ des Bezugsobjekts für alle Verweise (Typbeschreibung auch für Schachteln im Keller, nicht möglich bei polymorphen Verweisen)

6.1 Methodik

1. Markiere alle benötigten Objekte:
   Spannende Bäume ausgehend von Verweisen im Keller (root pointers)
2. Alternativen:

(a) freien Platz in Freilisten sammeln

(b) Kompression: benötigte Objekte im Speicherbereich zusammenschieben

(c) Kopieren: benötigte Objekte in neuen Speicherbereich kopieren

(b), (c) außerdem Adressen korrigieren

Voraussetzung:

- alle Verweise auffindbar
- bei Kompression, Kopieren: kein Nicht-Verweis wird fehlerhafterweise als Verweis interpretiert, sonst wird z.B. eine Gleitpunktzahl geändert.

6.1 Probleme

- Länge der Objekte alle gleich? Sonst: woher bekannt?
- Erkennen von Verweisen? (Register, Keller, Register, Halde)
- Verweise immer auf Objektanfang?
- Kellerlänge bei Tiefensuche? (keine Rekursion, expliziter Keller oder Zeigerumkehr)
- Lokalität der Objekte?

6.1 Aufwand

- Programmlänge der Speicherbereinigung ca. 300-500 (Maschinen-) Befehle

H Haldenlänge, R Umfang des noch benötigten Restes, $H - R$ gewonnener Platz

- Aufwand $c_1 R + c_2 H$ (für Markieren und Aufsammeln)

- Relativer Aufwand $\frac{c_1 R + c_2 H}{H - R}$ (in # Befehlszyklen/Wort)
- $H / R > 0.5$: H vergrößern
- $H \geq$ verfügbarer Hauptspeicher:
  Transporte Haupt-/Hintergrundspeicher dominieren Kosten
- **Anzahl Pufferfehler (Cachemiss) dominieren die Kosten immer!**
  Auf Lokalität achten!

Kapitel 6: Speicherbereinigung

1. Einleitung
   - Alternativen
   - Methodik
2. Markieren
3. Verfahren
   - mit Freiliste
   - mark-and-sweep
   - mark-and-copy
   - Generationenspeicher
   - Paralleler Speicherbereiniger
4. Literatur

6.2 Markieren

Tiefensuche (mit explizitem Keller oder Zeigerumkehr) oder
Breitensuche (bei kopierender SB)

Voraussetzungen:

- Markierungsbit für jedes Objekt
- bei nicht-uniformen Objekten ($\neq LISP$): komplettes Typ-Layout oder
  bei polymorphen Verweisen: Typ-Layout + Typkennung jedes Objekts
- Umfang der Objekte muß bekannt sein
- bei Verweisen in ein Objekt:
  (Objektanfang, Relativadresse) benutzen
- Welche Register enthalten Verweise?
  - bei Prozeduraufruf Liste der Register mit Verweisen übergeben

6.2 Tiefensuche mit Zeigerumkehr

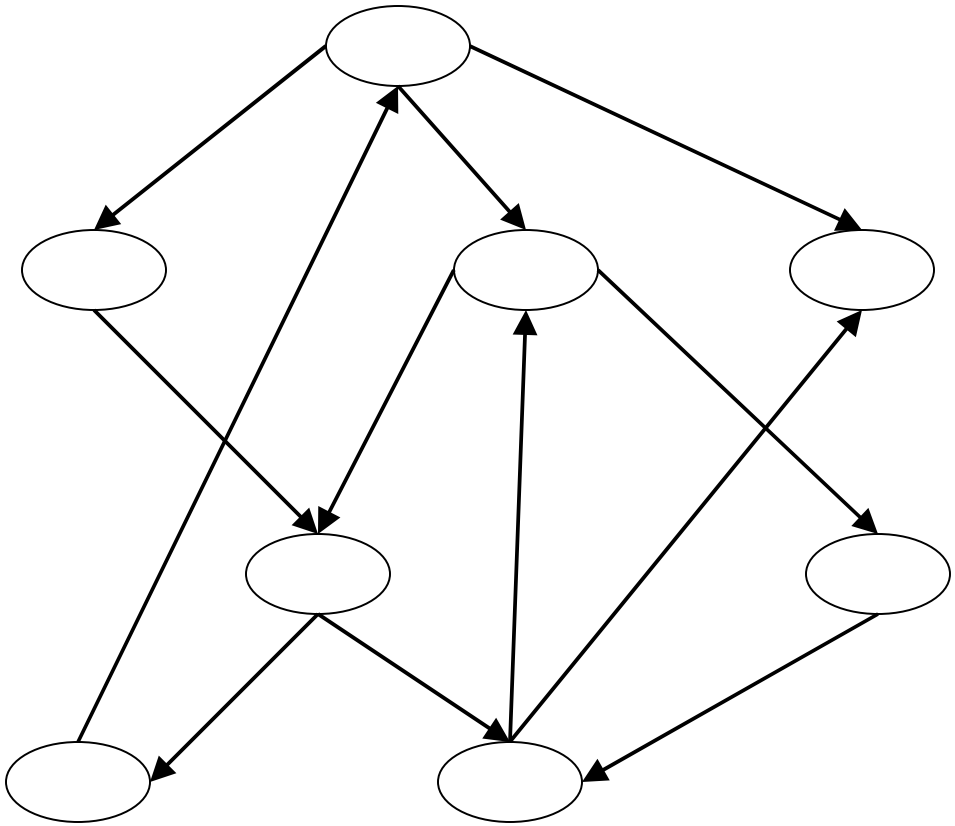
Tiefensuche zur Konstruktion eines spannenden Walds aller noch zugänglichen Objekte benötigt:

- eine boolesche Größe ,,markiert" für jedes Objekt (Markierungsbit): Kennzeichen bereits besuchter Objekte
- Durchlauf als Rekursion über alle erreichbaren Objekte? Keller kann überlaufen!

Alternative: Zeigerumkehr nach Schorr-Waite: vor Besuch eines Sohnes benutze den Verweis auf den Sohn, um auf den Vater zu zeigen, bei Rückkehr setze den alten Verweis wieder ein.

- Pfad wird im Graph gespeichert, kein Keller nötig
- zusätzlich in jeder Ecke Zähler nötig, welcher Sohn als nächster besucht werden soll
- Zählergröße = max. Anzahl Verweise im Objekt

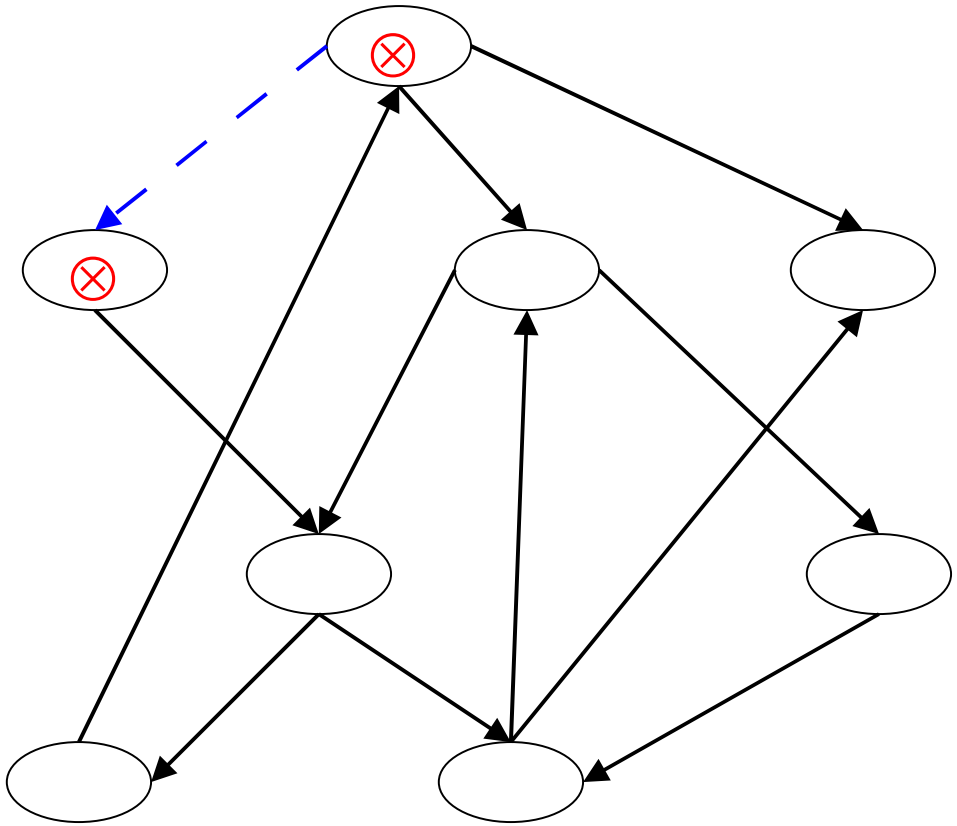
6.2 Zeigerumkehr nach Schorr-Waite



⊗: Markierung

*: Anzahl Söhne

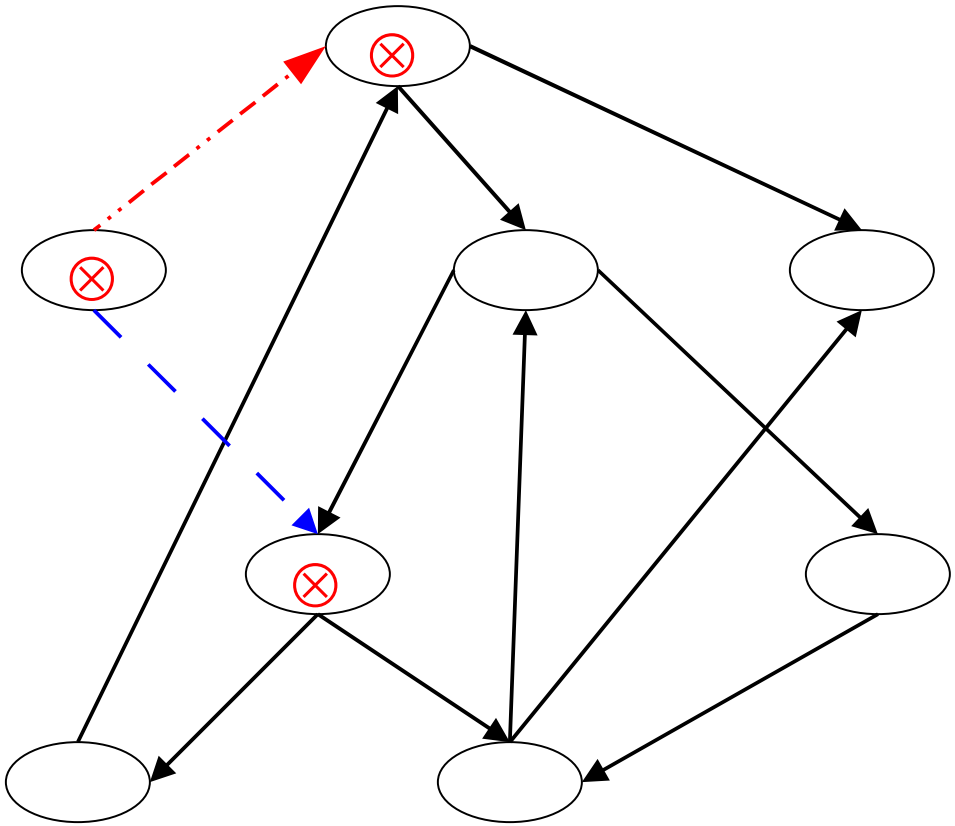
6.2 Zeigerumkehr nach Schorr-Waite



⊗: Markierung

*: Anzahl Söhne

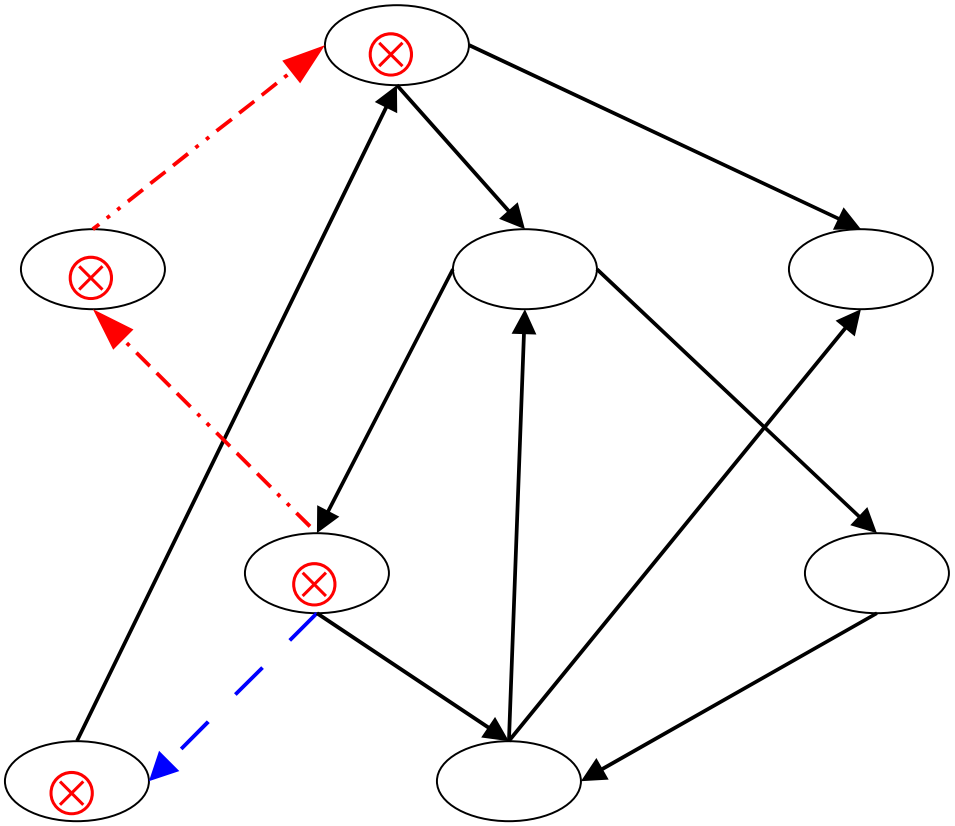
6.2 Zeigerumkehr nach Schorr-Waite



⊗: Markierung

*: Anzahl Söhne

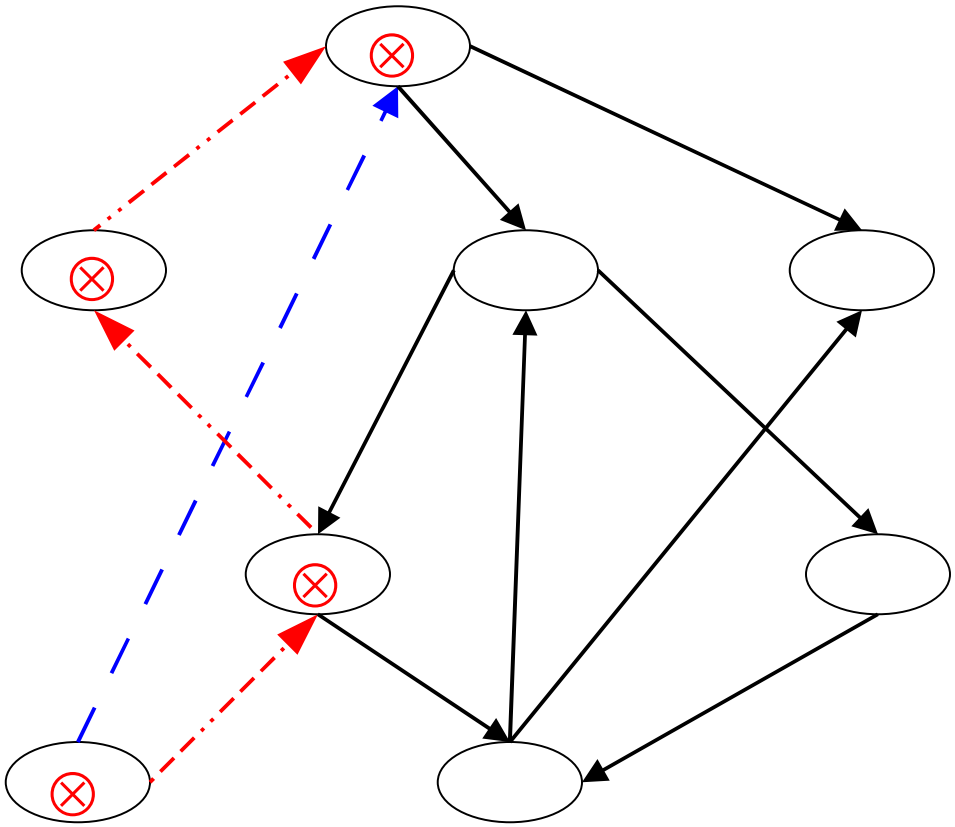
6.2 Zeigerumkehr nach Schorr-Waite



⊗: Markierung

*: Anzahl Söhne

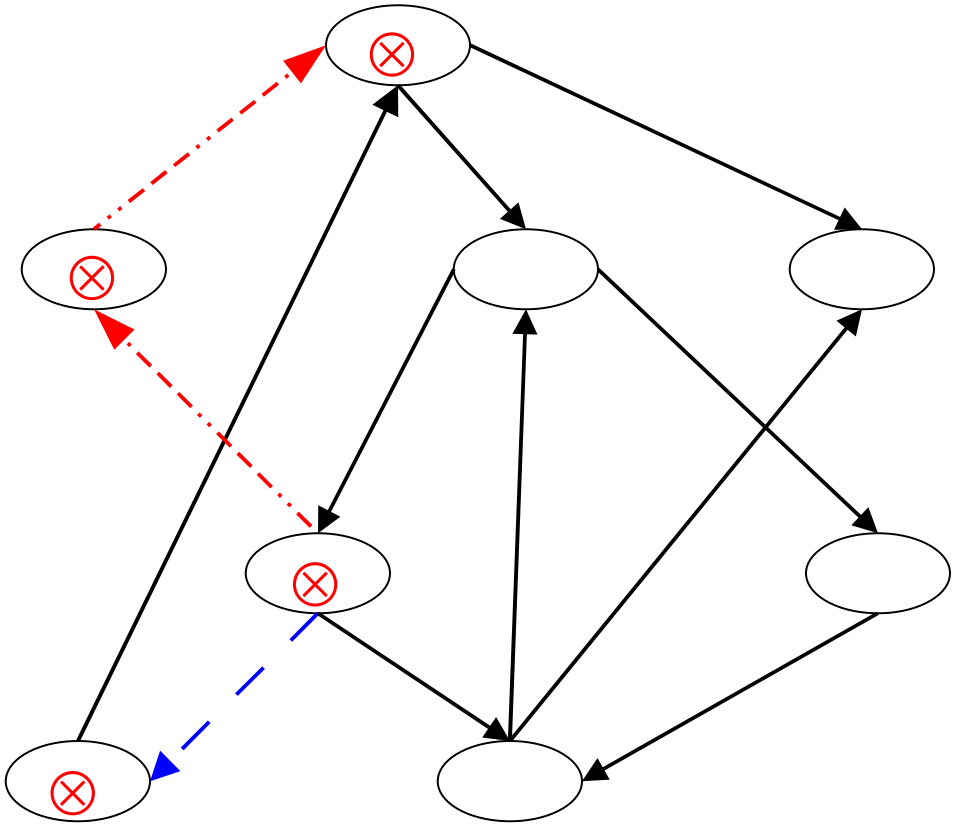
6.2 Zeigerumkehr nach Schorr-Waite



⊗: Markierung

*: Anzahl Söhne

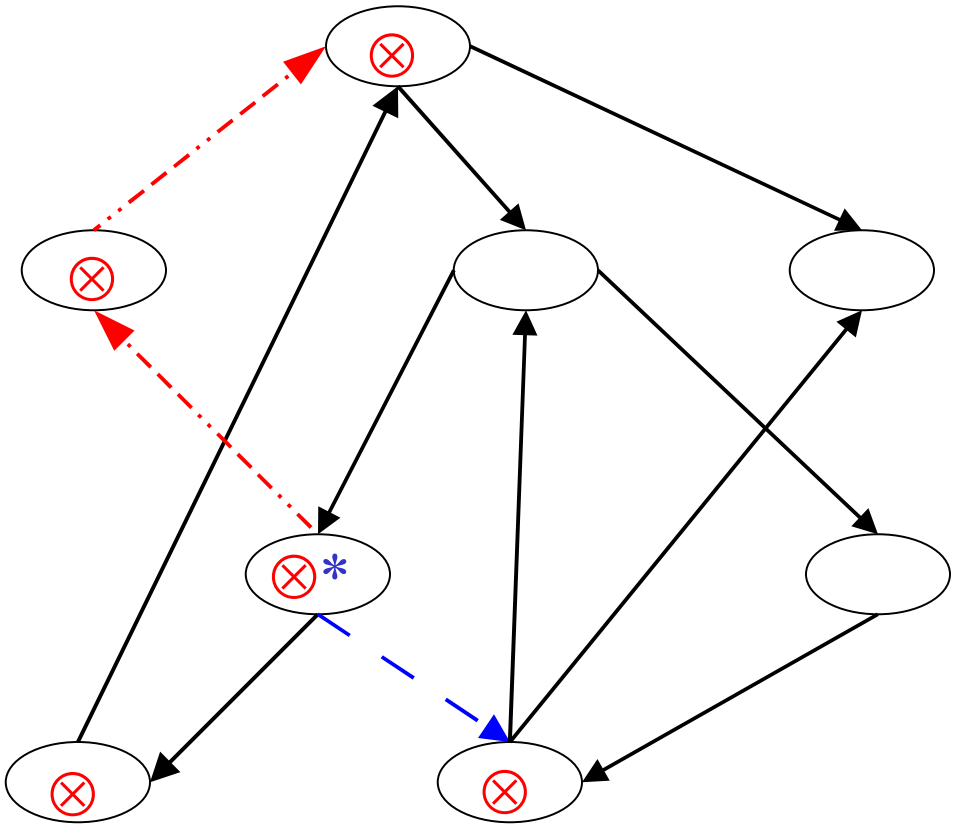
6.2 Zeigerumkehr nach Schorr-Waite



⊗: Markierung

*: Anzahl Söhne

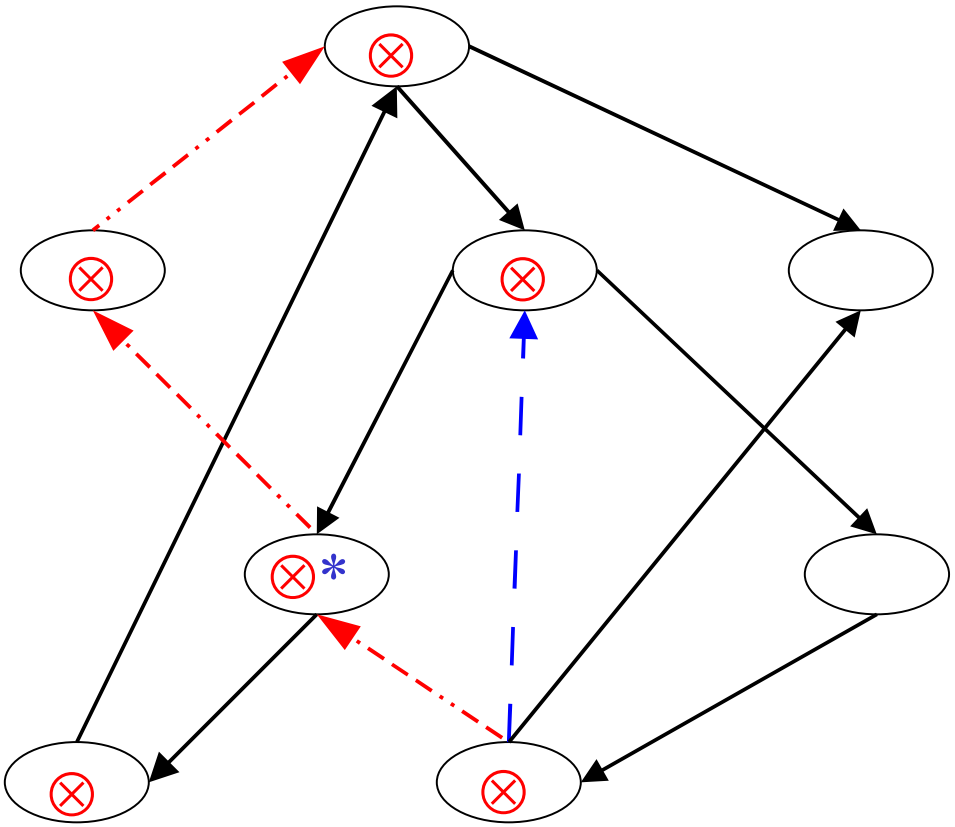
6.2 Zeigerumkehr nach Schorr-Waite



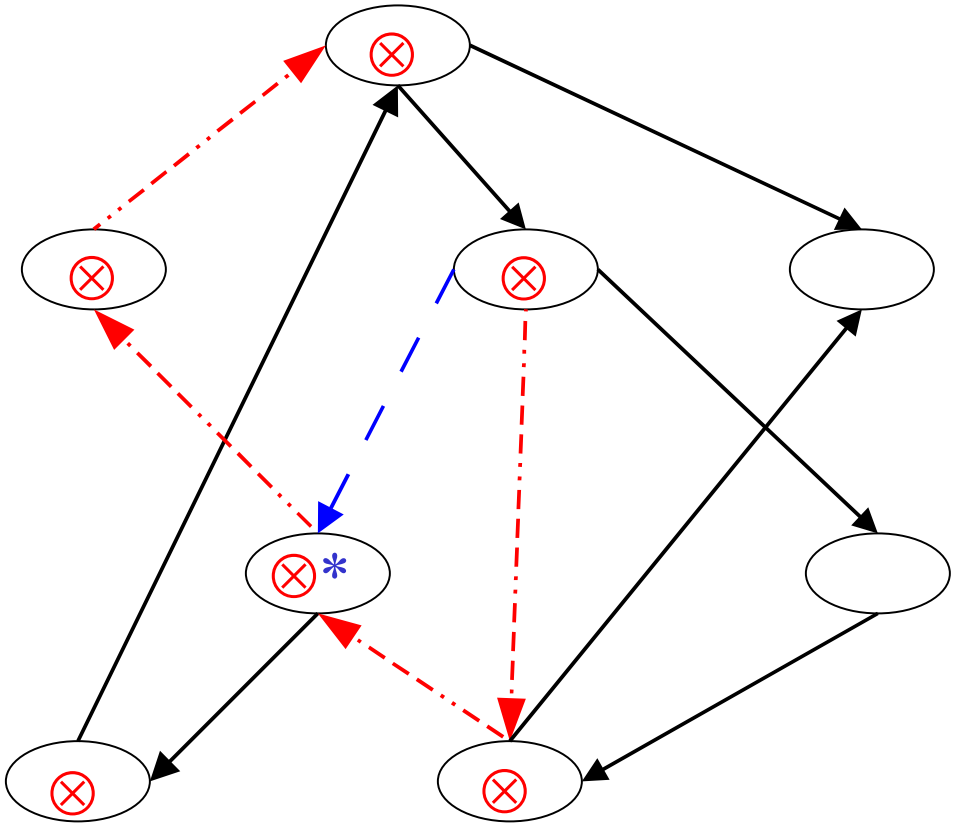
⊗: Markierung

*: Anzahl Söhne

6.2 Zeigerumkehr nach Schorr-Waite



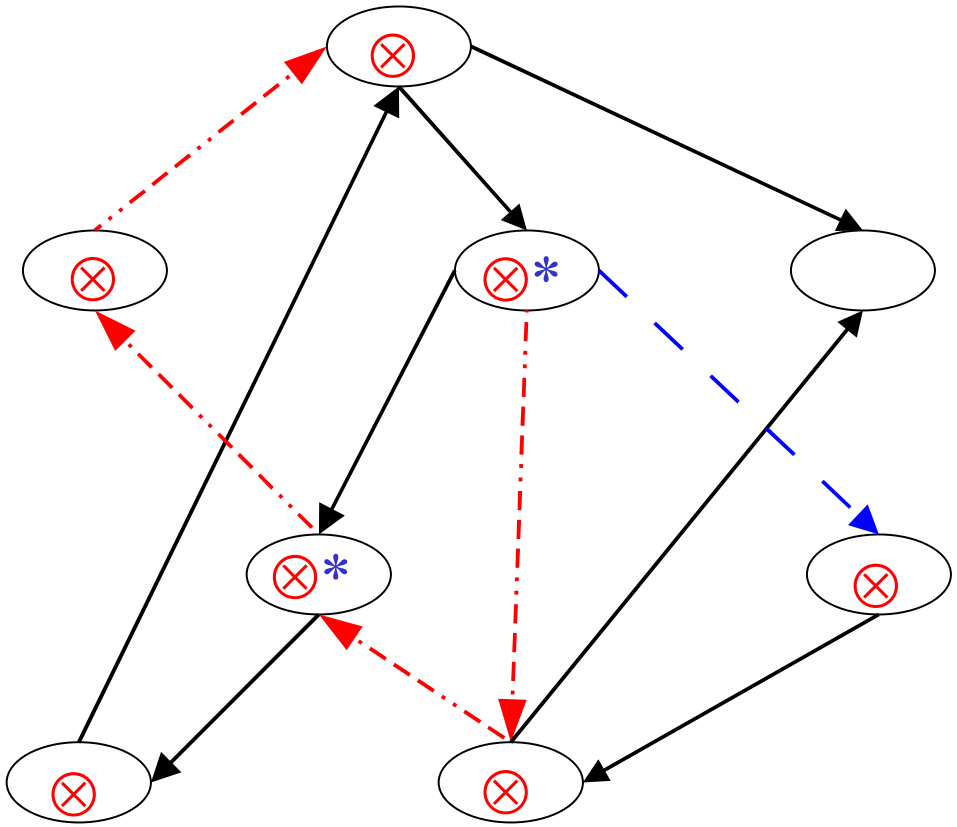
6.2 Zeigerumkehr nach Schorr-Waite



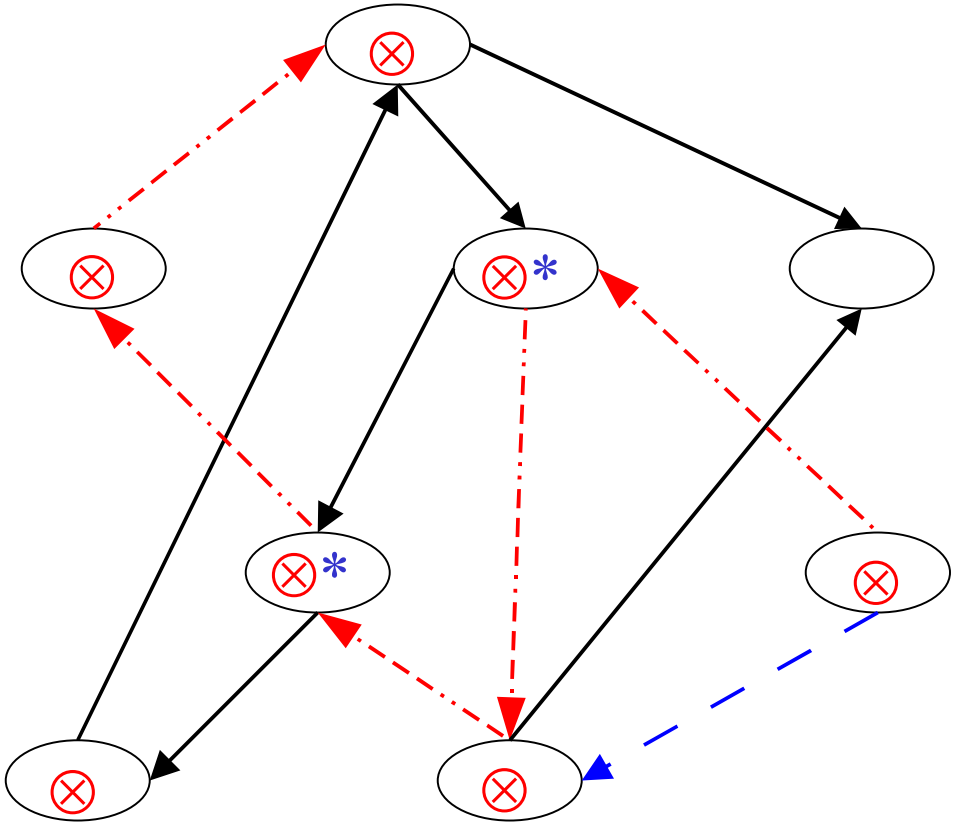
⊗: Markierung

*: Anzahl Söhne

6.2 Zeigerumkehr nach Schorr-Waite



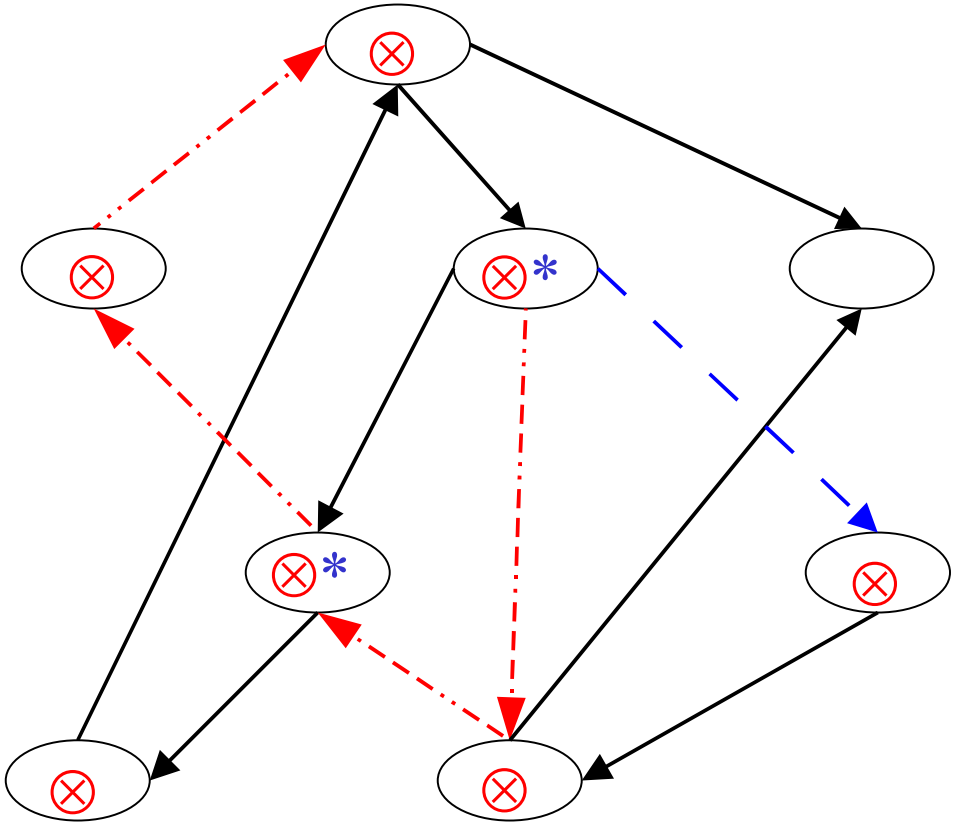
6.2 Zeigerumkehr nach Schorr-Waite



⊗: Markierung

*: Anzahl Söhne

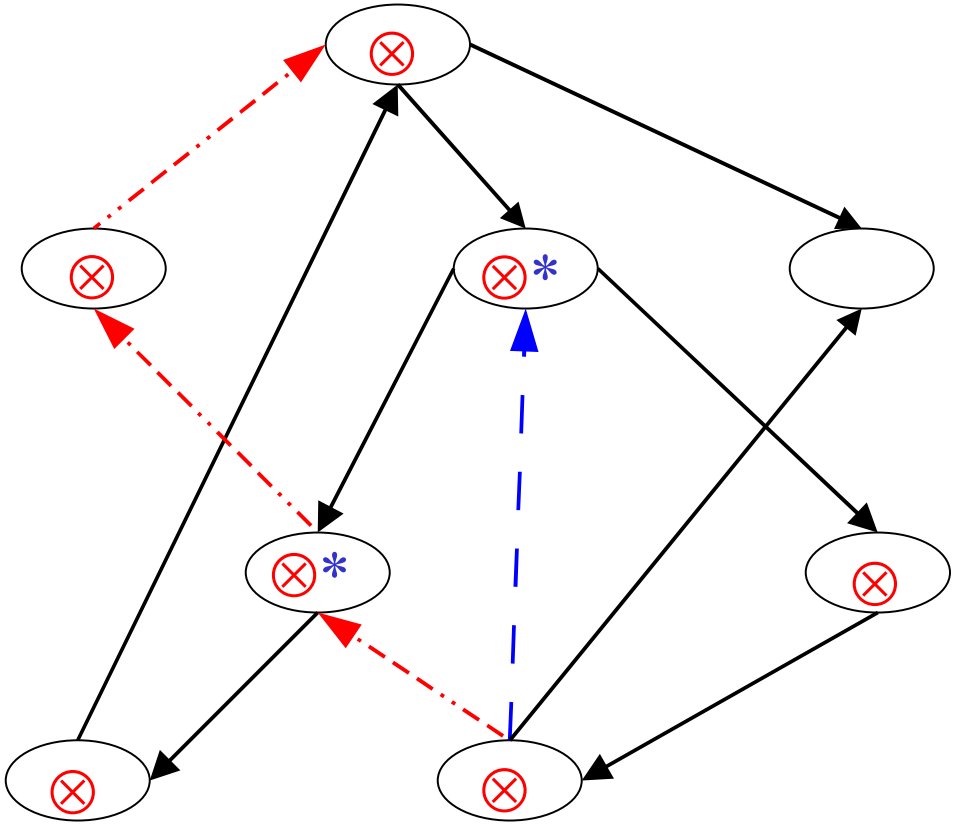
6.2 Zeigerumkehr nach Schorr-Waite



⊗: Markierung

*: Anzahl Söhne

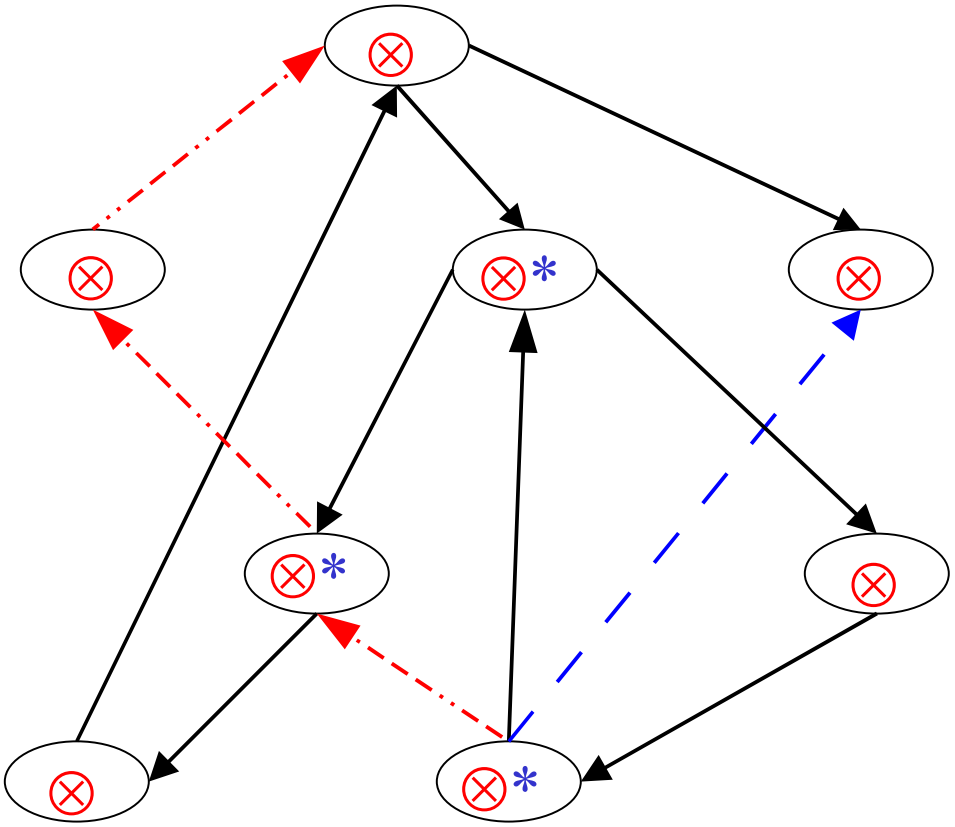
6.2 Zeigerumkehr nach Schorr-Waite



⊗: Markierung

*: Anzahl Söhne

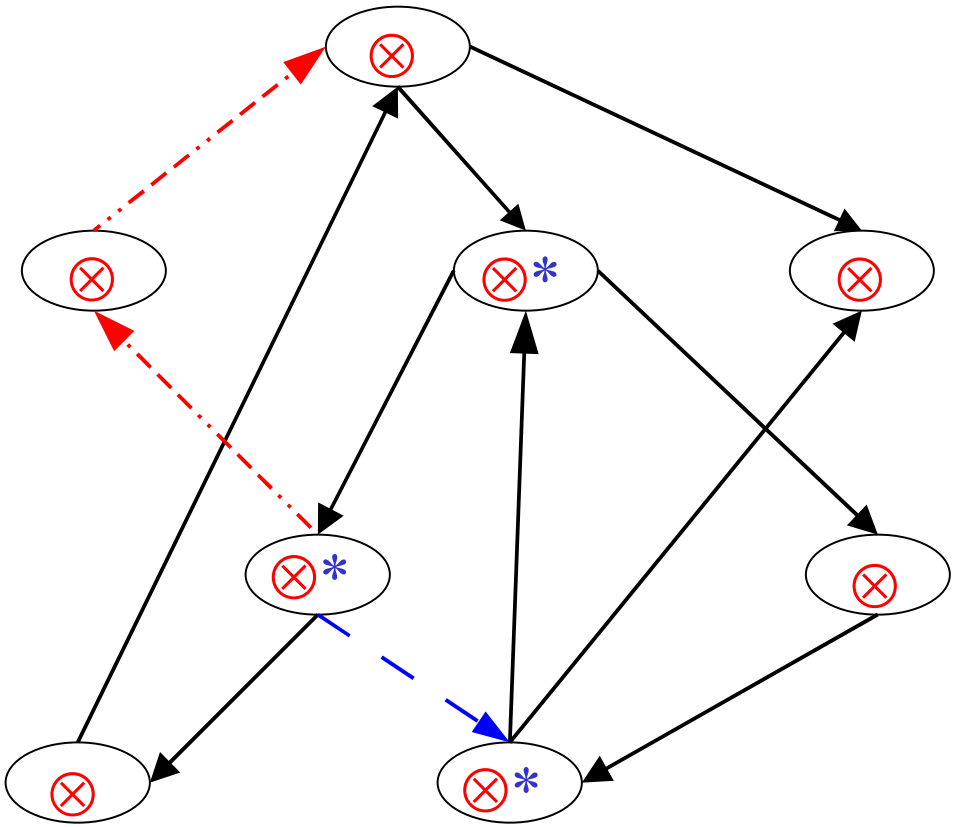
6.2 Zeigerumkehr nach Schorr-Waite



⊗: Markierung

*: Anzahl Söhne

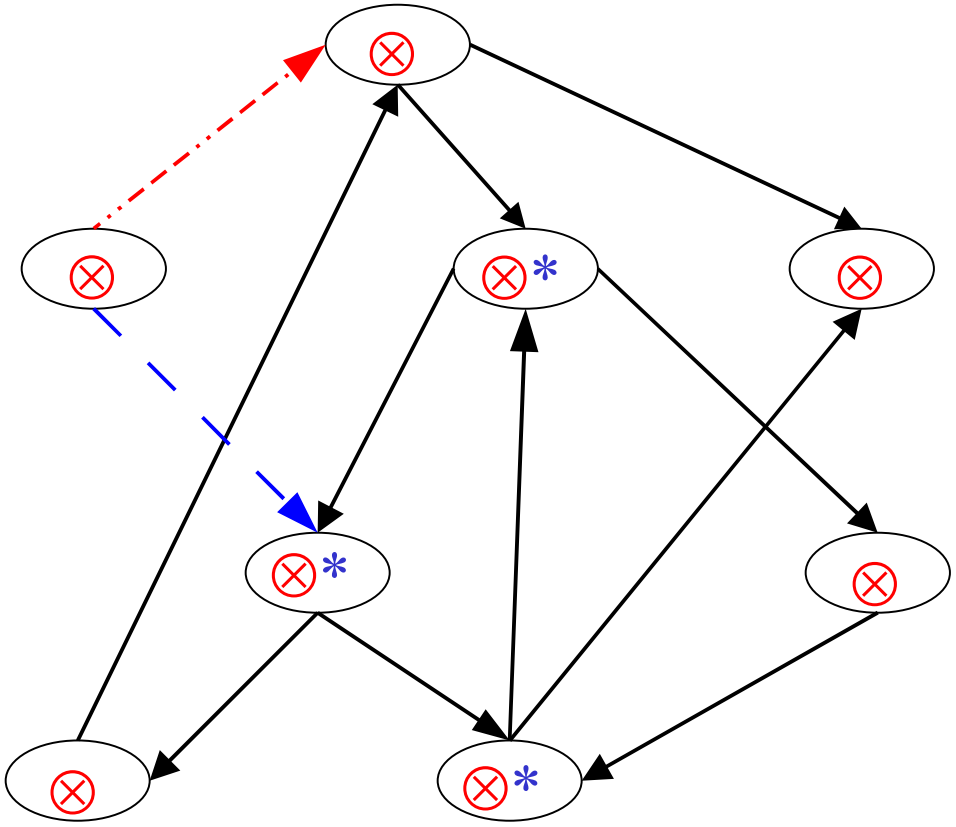
6.2 Zeigerumkehr nach Schorr-Waite



⊗: Markierung

*: Anzahl Söhne

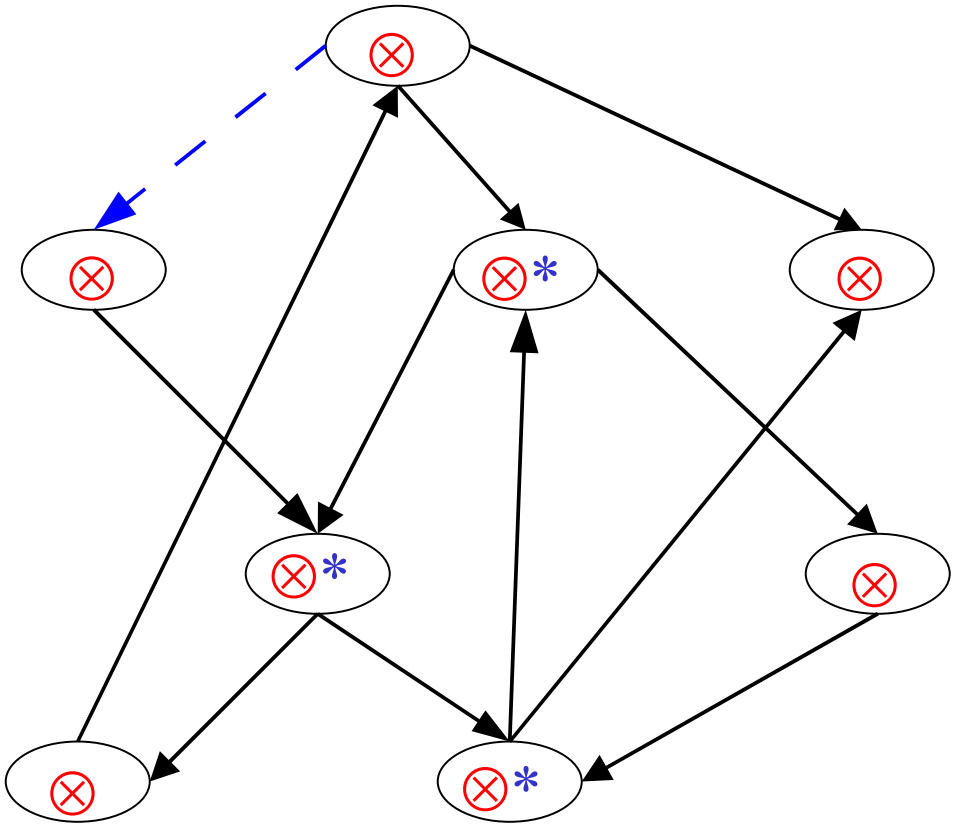
6.2 Zeigerumkehr nach Schorr-Waite



⊗: Markierung

*: Anzahl Söhne

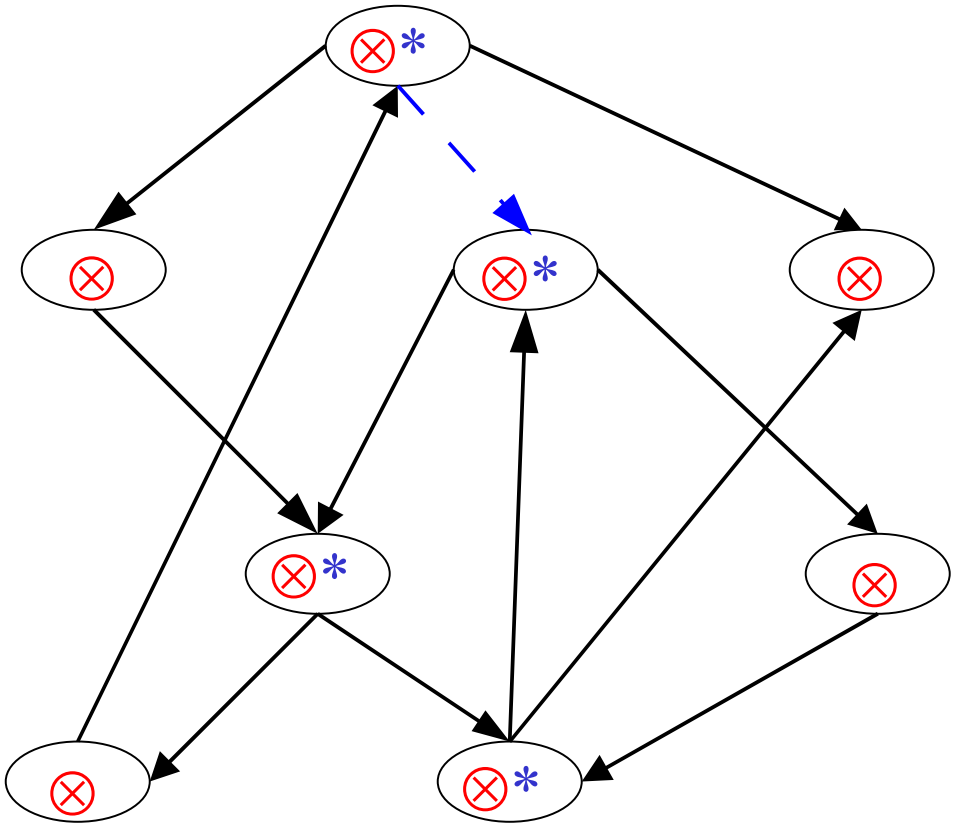
6.2 Zeigerumkehr nach Schorr-Waite



⊗: Markierung

*: Anzahl Söhne

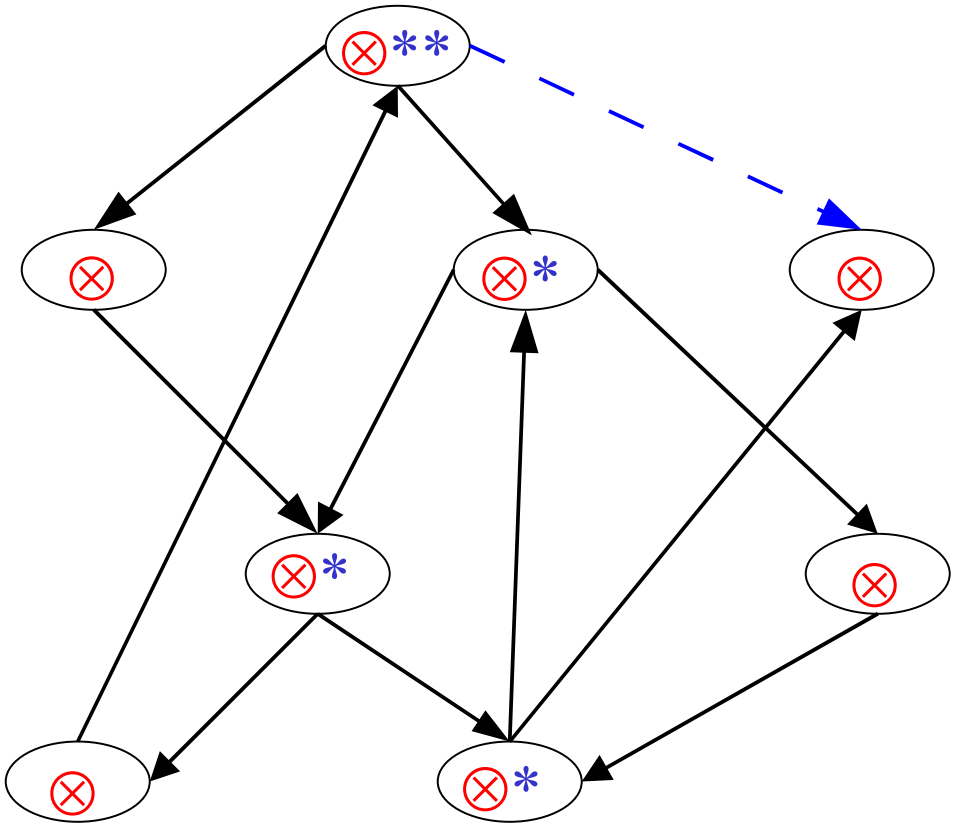
6.2 Zeigerumkehr nach Schorr-Waite



⊗: Markierung

*: Anzahl Söhne

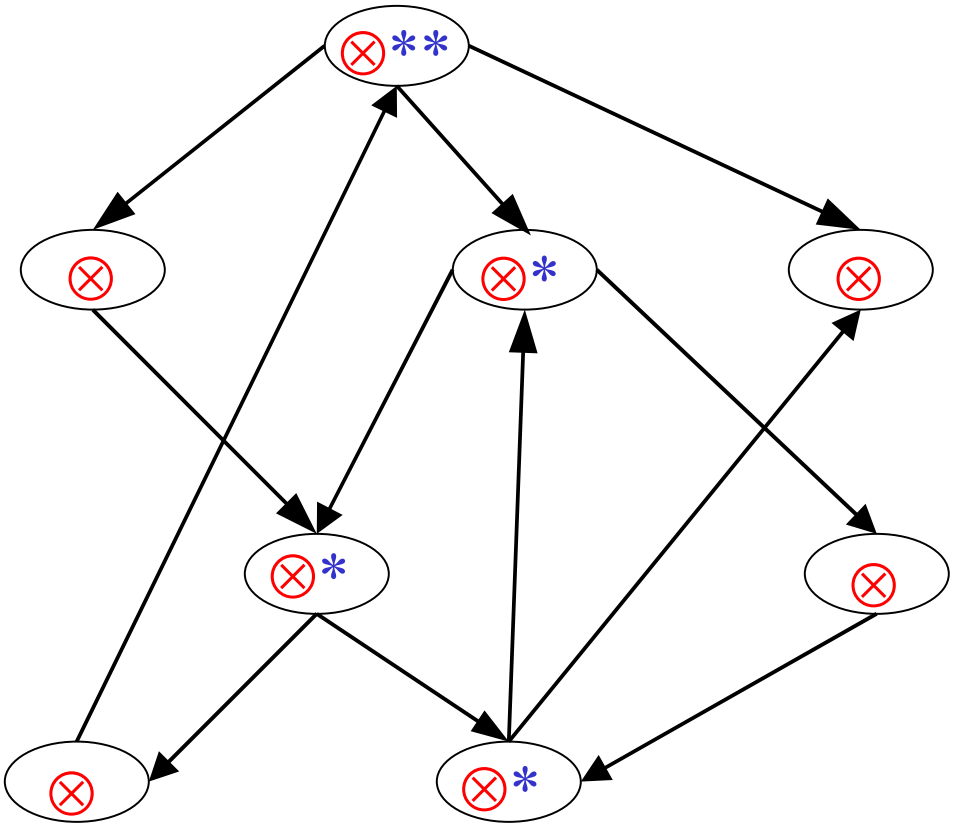
6.2 Zeigerumkehr nach Schorr-Waite



⊗: Markierung

*: Anzahl Söhne

6.2 Zeigerumkehr nach Schorr-Waite



⊗: Markierung

*: Anzahl Söhne

fertig!

6.2 Ergebnis der Markierung

- Durch Markieren sind alle noch benötigten Objekte bekannt
- Alternativen:
  - Durch lineares Absuchen der Halde sämtliche Lücken (= nicht mehr benötigte Objekte) ermitteln: $O(H)$
  - Aufsammeln der Lücken in Freiliste (eine, mehrere) kann bereits beim Markieren geschehen: $O(R)$
  - Kopieren der noch benötigten Objekte in neuen Speicherbereich
    - Kompression
    - kein Absuchen der Halde erforderlich: $O(R)$

Kapitel 6: Speicherbereinigung

1. Einleitung
   - Alternativen
   - Methodik
2. Markieren
3. Verfahren
   - mit Freiliste
   - mark-and-sweep
   - mark-and-copy
   - Generationenspeicher
   - Paralleler Speicherbereiniger
4. Literatur

6.3 SB mit Freiliste

- Aufsammeln der Speicherlücken
- Zusammenschließen benachbarter Zellen

- Freilisten nach *Baker*: *alle* Objekte bei Allokation in Liste aufnehmen; markierte Objekte in neue Liste
  - Vorteil: nach der Markierung ist die neue Freiliste unmittelbar bekannt
  - Nachteil: Platzbedarf doppelte Verkettung
  - Nachteil: Freiliste nicht nach aufsteigenden Adressen sortiert, kein Zusammenschluß von Lücken (externe Fragmentierung)

- genereller Vorteil von Freilisten: Objekte ändern ihre Adresse nicht
- Nachteil: Probleme bei neuen großen Objekten

6.3 SB mit Kompression (*mark-and-sweep*)

Drei Durchgänge durch den Speicher (Aufwand $O(H + R)$)

1. Markieren, Lücken bestimmen,
2. Adressen ändern
   mit erneuter Tiefensuche
3. Objekte zusammenschieben,
   Markierung löschen

- Bittabelle zur Lücken-bestimmung oder
  (*Summe Lückenlängen, aktuelle Lückenlänge, Beginn nächste Lücke*) am Lückenanfang eintragen

Nachteil: Aufwand zur Berechnung neuer Adressen

6.3 Adressen ändern vor Kompression

Verfahren nach Wegbreit, 1974

bei Eintrag

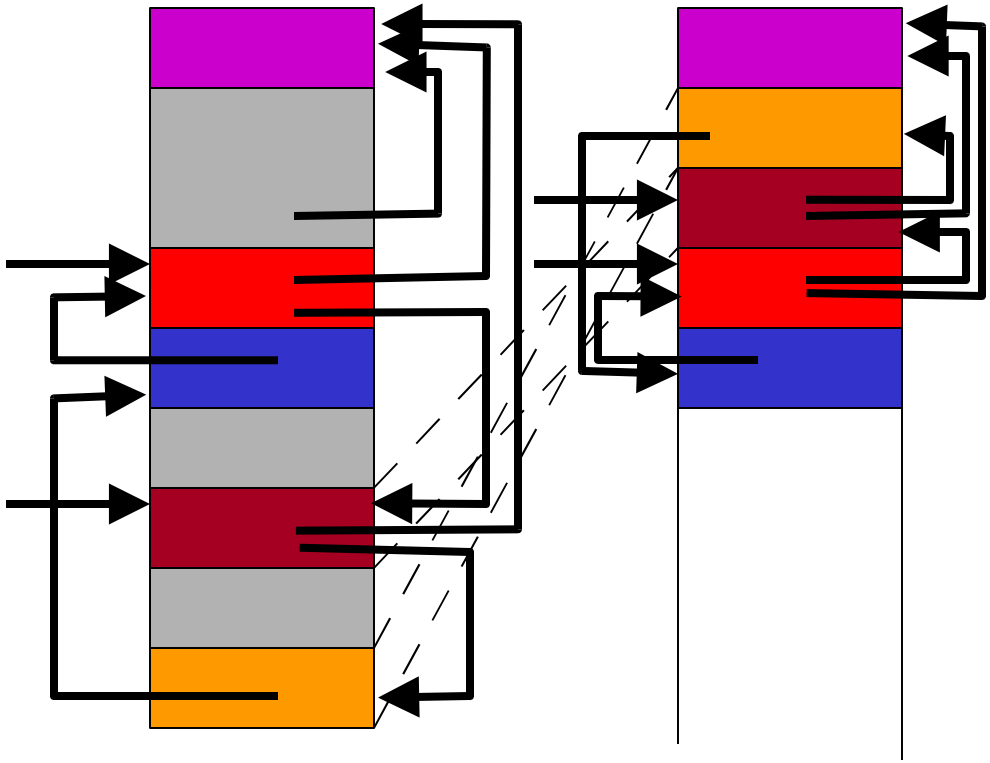
(*Summe Lückenlängen, aktuelle Lückenlänge, Beginn nächste Lücke*)

am Lückenanfang:

- vor dem Umzug eines Objekts o der Länge l jeden Verweis $adr(o)$ durch $adr(o)$ - *Summe Lückenlängen* ersetzen, Summe aus Adresse der letzten Lücke entnehmen
- nach dem Umzug nicht möglich, weil aus altem Verweis nicht auf neue Adresse geschlossen werden kann

6.3 Kompression bei Objekten gleicher Länge

- Umzug von hinten nach vorn möglich,
- weniger Kopieroperationen
- alter Ort wird frei
  - kann Nachsendeadresse aufnehmen
  - dadurch Adreßänderung nach Umzug möglich
- Aber: teurer als Freiliste
- Berücksichtige aber auch:
  - Größe *working set*
  - Lokalität

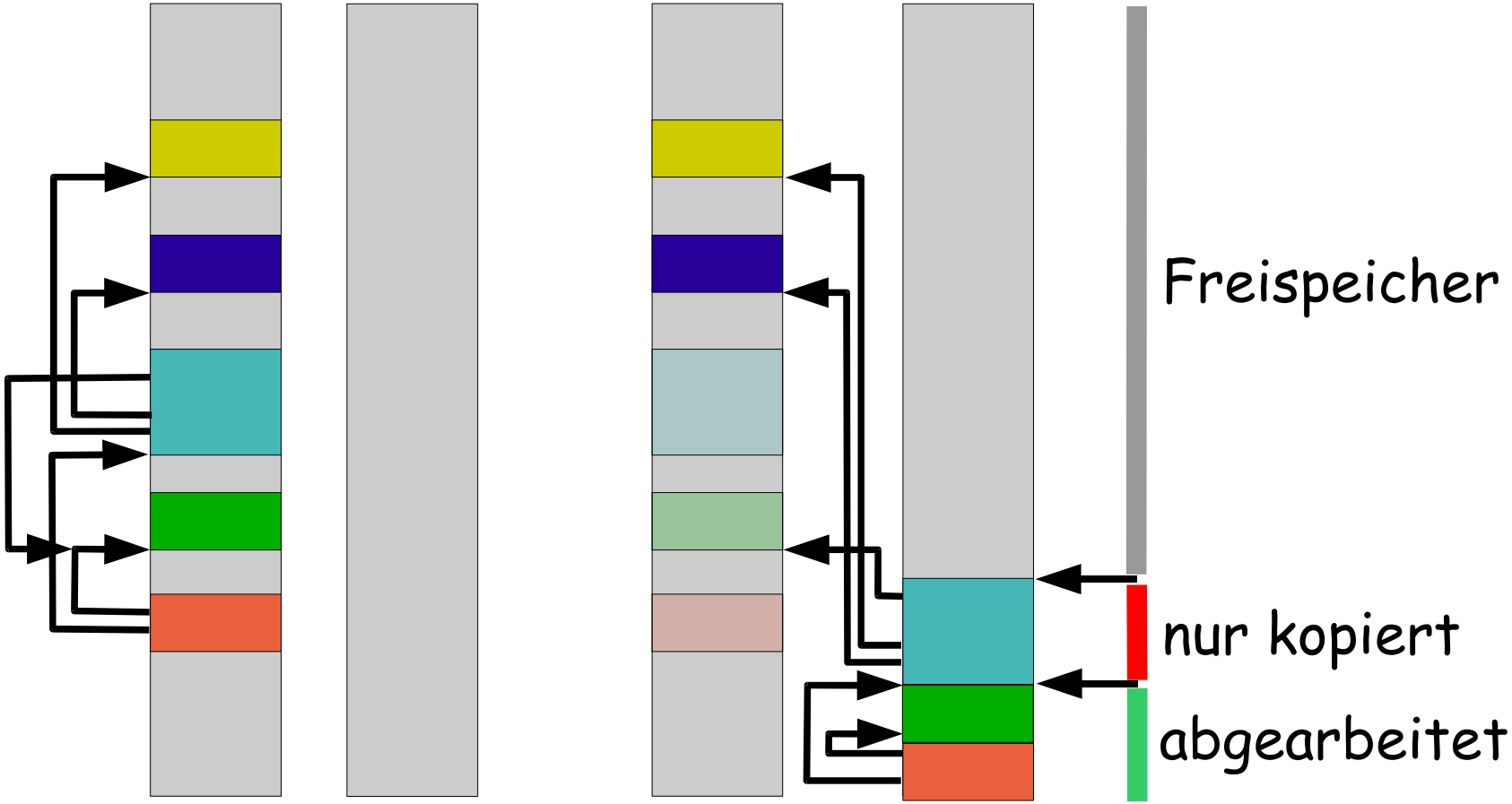


6.3 Kopieren (*mark-and-copy*)

Unterhalte zwei Speicherbereiche *(from-Space, to-Space)*

- Traversiere from-Space
  - kopiere gefundene Objekte nach to-Space
  - hinterlasse Nachsendeadresse in kopierten Objekten
- Objekte werden sequentiell im Speicherbereich angeordnet
  - Nutze Zeiger in to-Space als Merker für noch zu kopierende Objekte (Cheney, 1970)
- Vorteil: Aufwand $O(R)$
  - sehr billige Allokation von Objekten
  - Breitensuche erhöht Lokalität: zusammengehörige Objekte in neuem Speicher benachbart
  - keine externe Fragmentierung im Speicherbereich
- Nachteil:
  - Verdopplung des virtuellen (!) Speicheraufwands
  - Zeigerfreie Objekte müssen kopiert werden (typ. 30% aller Objekte)

6.3 Kopieren (*mark-and-copy*)



6.3 Generationenspeicher (*generation scavenging*)

Beobachtung: Neue Objekt sterben schnell oder werden alt;
alte Objekte sterben selten.

Idee:

- Partitioniere Speicher in n Partitionen $G_0, G_1, \ldots, G_{n-1}$, z.B. $n = 2$ oder 3
- Speicherzuteilung nur in G_{n-1}
- Bei Überlauf SB mit Verweisen in Keller und $G_0, G_1, \ldots, G_{n-2}$ als Anker
- Benötigte Objekte altern, d. h., sie wandern von G_{n-1} nach G_{n-2}, usw., also SB mit Kopieren außer in G_0
- Häufige SB in G_{n-1}, seltener in G_{n-2}, noch seltener in $G_{n-3}\ldots$
- Tricks um explizites Durchsuchen von $G_0, G_1, \ldots, G_{n-2}$ zu vermeiden: Vormerklisten, Vormerkseiten

6.3 Parallele Speicherbereinigung (Dijkstra: *garbage collection on the fly*)

Speicherbereiniger (*collector*) parallel zum Programm (*mutator*)

- Dreifarbenmarkierung: weiß (noch nicht besucht) grau (besucht, Tiefensuche noch nicht fertig) schwarz (samt allen Kindern besucht)
  - Zuweisung färbt Bezugsobjekt grau (nur dann wenn es weiß ist), sonst keine Intervention des Programms
  - Invarianten garantiert durch SB, nicht gestört durch Programm:
    - Schwarz zeigt nicht auf weiß
    - Grau ist in Bearbeitung
    - Nach Ende Markierungsphase sind alle weißen Objekte überflüssig
- Aufwand: Synchronisation bei Bildung neuer Objekte

  Befehl zum bedingten Graufärben
- Vorteil: keine längere Unterbrechung des Programms (inkrementelle SB)

6.3 Anforderungen an Übersetzer

- Alle Objekte vom gleichen Typ: keine Anforderungen
- Sonst:
  Komplette Typ-Layouts zur Laufzeit verfügbar machen,
  Erkennen von Verweistypen:
  1. Statisch aus Layout (nur bei monomorphen Typen möglich)
  2. Dynamisch aus Typkennung des Bezugsobjektes (zusätzliches Kennfeld)
- Problem: Verweise in Registern und Registerablagen
- Information ist Teilmenge der Information für Testhilfen

6.4 Literatur

[1] Andrew *W.* Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998, Kapitel 13.

[2] Hans-Jürgen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice Experience*, 18(9): 807-820, 1988. In Sather verwandter Speicherbereiniger

[3] Edsgar *W.* Dijkstra, Leslie Lamport, *A. J.* Martin, *C. S.* Scholten, and *E. F. M.* Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM,* 21(11): 966-975, November 1978. Parallele Speicherbereinigung

[4] *R.* Jones and *R.* Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996. Sehr gute Übersicht

[5] David Ungar. Generation scavenging: a nondisruptive high performance storage reclamation algorithm. *SIGPLAN Notices*, 19(5): 157-167, 1984.

[6] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. *SIGPLAN* Notices, 23(11): 1-17, 1988.

[7] Paul *R.* Wilson. Uniprocessor garbage collection techniques. In Bekker and Cohen: Memory Management, *LNCS* 637, 1992, pages 1-42. Übersichtsartikel

[8] Fridtjof Siebert. Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages. Dissertation, Universität Karlsruhe, Fakultät für Informatik, 2001, AICAS GmbH, ISBN: 3-8311-3893-1