Dijkstra's Algorithm with Fibonacci Heaps: An Executable Description in CHR

Jon Sneyers, Tom Schrijvers, Bart Demoen
K.U.Leuven, Belgium

WLP, Vienna, February 22-24, 2006

驰

Overview

Overview

Constraint Handling Rules [Frühwirth 1991]

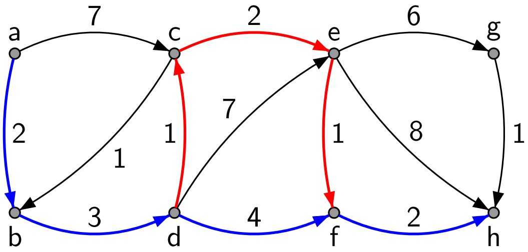
- High-level language extension
- Multi-headed committed-choice guarded rules
- Originally designed for constraint solvers
- General-purpose programming language
- Every algorithm can be implemented with the optimal time and space complexity! [Sneyers-Schrijvers-Demoen CHR'05]

Very nice, but...

- Can all algorithms be implemented
  in a natural, elegant, compact way?
- Some empirical evidence
  e.g. union-find [Schrijvers-Frühwirth TPLP 2006]
- and: What about **constant factors**?

Overview

The single-source shortest path problem



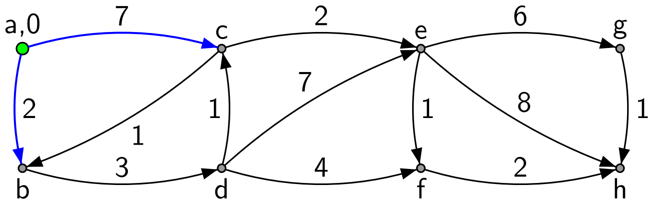
- Important problem in algorithmic graph theory
- Given: a weighted directed graph and a source node
- Wanted: the distance from the source to all other nodes (distance: total weight of a shortest path)
- If the weights are non-negative: Dijkstra's algorithm

Representation

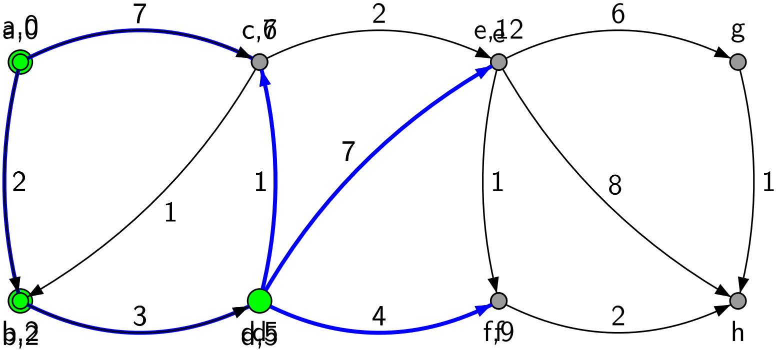
- Edge from A to B with weight W: `edge(A,B,W)`
- Weights: numbers > 0
- Node names: integers in $[1, n]$ (number of nodes: n)
- Query: `edge/3`'s followed by `dijkstra(S)` where `S` is the source node
- Output: `distance(X,D)`'s meaning "the distance from the source node `S` to the node `X` is `D`"

Dijkstra's Algorithm [Dijkstra 1959]

- During algorithm, nodes can be unlabeled, labeled or scanned
- Initially: all nodes unlabeled, except source which gets label 0
- Node `X` is scanned if there is a `distance(X,_)` constraint
- We start by scanning the source:
  `dijkstra(A) <=> scan(A,0).`



Dijkstra's Algorithm



Scanning a node

- Scanning a node: first make it scanned
  `scan(N,L) ==> distance(N,L).`
- Then label its neighbours:
  `scan(N,L), edge(N,N2,W) ==> relabel(N2,L+W).`
- Finally, pick the next node to scan. Pick a labeled node with the smallest label:
  `scan(N,L) <=> extract_min(N2,L2) | scan(N2,L2).`
- If there is no next node, stop:
  `scan(N,L) <=> true.`

Relabeling a node

- (re)labeling a node: do nothing if it is already scanned
  `distance(N,_) \ relabel(N,_) <=> true.`
- Otherwise, add or decrease its label:
  `relabel(N,L) <=> decr_or_ins(N,L).`
- Still need to define `decr_or_ins/2` and `extract_min/2`

Priority queues

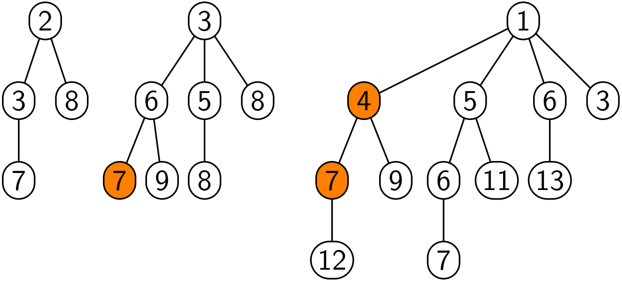
- Store (item,key) pairs (item=node, key=tentative distance)
- `extract_min/2` gives the pair with the minimal key and removes it from the queue
- `ins/2` adds a pair
- `decr/2` updates the key for some item **if** the new key is smaller than the original
- `decr_or_ins/2` adds the pair if it is not in the queue, decreases its key otherwise

Simple priority queues

- Sorted list: `extract_min/2` in $O(1)$, `decr_or_ins/2` in $O(n)$
  $\longrightarrow$ Dijkstra in $O(mn)$ (m edges, n nodes)
- Array: `extract_min/2` in $O(n)$, `decr_or_ins/2` in $O(1)$
  $\longrightarrow$ Dijkstra in $O(n^2)$
- Binary heap: `extract_min/2` and `decr_or_ins/2` in $O(\log n)$
  $\longrightarrow$ Dijkstra in $O(m \log n)$

Overview

Fibonacci Heaps [Fredman-Tarjan 1987]



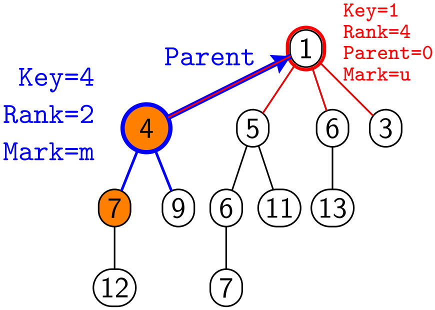
- Advanced priority queue
- `extract_min/2` in $O(\log n)$, `decr_or_ins/2` in $O(1)$
  $\longrightarrow$ Dijkstra in $O(m + n \log n)$
- Optimal for Dijkstra-based shortest path!

CHR representation of F-Heaps

- Store the pairs as `item/5` constraints:

  `item(Item,Key,Rank,Parent,Mark)`

- Parent is 0 if the pair is a root, > 0 otherwise
- Rank $=$ number of children



Fibonacci Heaps in CHR

- Maintain the current minimal pair:
  ```
  min(_,A) \ min(_,B) <=> A =< B | true.
  ```
- Heap-ordered trees: parent has smaller key than children
  → minimum must be a root
- No two roots can have the same rank:
  ```
  item(A,K1,R,0,_), item(B,K2,R,0,_) <=> K1 =< K2 |
    item(A,K1,R+1,0,u), item(B,K2,R,I1,u).
  ```

Fibonacci Heap operations

- Insert is easy: add new root pair and candidate minimum

```
insert(I,K) <=> item(I,K,0,0,u), min(I,K).
```

- Extract minimum: remove, children2roots, find new minimum

```
extract_min(X,Y), min(I,K), item(I,_,_,_,_)
  <=> ch2rt(I), findmin, X=I, Y=K.
extract_min(_,_) <=> fail.
```

- Children2roots:

```
ch2rt(I) \ item(C,K,R,I,_) <=> item(C,K,R,0,u).
ch2rt(I) <=> true.
```

- Find new minimum: only search roots!

```
findmin, item(I,K,_,0,_) ==> min(I,K).
findmin <=> true.
```

Decrease-key-or-insert

- New key smaller: decrease key

```
item(I,O,R,P,M), decr_or_ins(I,K)
  <=> K<O | decr(I,K,R,P,M).
```

(note: `item/5` is removed, `decr/5` will re-insert it)

- New key bigger: do nothing

```
item(I,O,_,_,_) \ decr_or_ins(I,K)
  <=> K >= O | true.
```

- No such item in the queue: insert

```
decr_or_ins(I,K) <=> insert(I,K).
```

That's (almost) it!

- Extremely compact, readable program: just 19 rules
- Pseudo-code descriptions of Fibonacci Heaps are usually longer! (and not executable)
- E.g. C implementation takes $>$ 300 lines, hard to understand/modify
- What about the performance of this program?

Comparison: SPLIB implementation in C


```
typedef struct arc_st{long len;struct node_st *head;}arc;typedef struct node_st{arc *first;long dist;struct node_st *parent;
struct node_st *heap_parent;struct node_st *son;struct node_st *next;struct node_st *prev;long deg;int status;int temp;}node;
#define BASE 1.61803 #define OUT_OF_HEAP 0 #define VERY_FAR 1073741823 #define NODE_IN_FHEAP(node)(node->status>OUT_OF_HEAP)
#define nod(node) (long)(node-nodes+1) #define MARKED 2 #define IN_HEAP 1 #define NNULL (node*)NULL #define NOT_ENOUGH_MEM 2
typedef struct fheap_st{node *min;long dist;long n;node **deg_pointer;long deg_max;}f_heap;f_heap fh;node *after,*before,
*father,*child,*first,*last,*node_c,*node_s,*node_r,*node_n,*node_l;long dg;void Init_fheap(n)long n;{fh.deg_max=(long)(
log((double) n)/ log(BASE)+ 1);if((fh.deg_pointer=(node**) calloc(fh.deg_max,sizeof(node*)))==(node**)NULL)exit(
NOT_ENOUGH_MEM);for(dg=0;dg<fh.deg_max;dg ++)fh.deg_pointer[dg]=NNULL;fh.n =0;fh.min=NNULL;} void Check_min(nd) node *nd;
{if(nd->dist<fh.dist){fh.dist=nd->dist;fh.min=nd;}} void Insert_after_min(nd) node *nd;{after=fh.min->next;nd->next=after;
after->prev=nd;fh.min->next=nd;nd->prev=fh.min;Check_min(nd);} void Insert_to_root(nd) node *nd;{nd->heap_parent=NNULL;nd->
status=IN_HEAP;Insert_after_min(nd);} void Cut_node(nd,father) node *nd,*father;{after=nd->next;if(after != nd){before=nd->
prev;before->next=after;after->prev=before;}if(father->son==nd)father->son=after;(father->deg)--;if(father->deg==0)father->
son=NNULL;}void Insert_to_fheap(nd) node *nd;{nd->heap_parent=NNULL;nd->son=NNULL;nd->status=IN_HEAP;nd->deg=0;if(fh.min==
NNULL){nd->prev=nd->next=nd;fh.min=nd;fh.dist=nd->dist;}else Insert_after_min(nd);fh.n ++;} void Fheap_decrease_key(nd) node
*nd;{if((father=nd->heap_parent)== NNULL)Check_min(nd);else{if(nd->dist<father->dist){node_c=nd;while(father != NNULL){
Cut_node(node_c,father);Insert_to_root(node_c);if(father->status==IN_HEAP){father->status=MARKED;break;}node_c=father;father
=father->heap_parent;}}}} node* Extract_min(){node *nd;nd=fh.min;if(fh.n>0){fh.n --;fh.min->status=OUT_OF_HEAP;first=fh.min
->prev;child=fh.min->son;if(first==fh.min)first=child;else{after=fh.min->next;if(child==NNULL){first->next=after;after->prev
=first;}else{before=child->prev;first->next=child;child->prev=first;before->next=after;after->prev=before;}}if(first!=NNULL)
{node_c=first;last=first->prev;while(1){node_l=node_c;node_n=node_c->next;while(1){dg=node_c->deg;node_r=fh.deg_pointer[dg];
if(node_r==NNULL){fh.deg_pointer[dg]=node_c;break;}else{if(node_c->dist<node_r->dist){node_s=node_r;node_r=node_c;} else
node_s=node_c;after=node_s->next;before=node_s->prev;after->prev=before;before->next=after;node_r->deg ++;node_s->
heap_parent=node_r;node_s->status=IN_HEAP;child=node_r->son;if(child==NNULL)node_r->son=node_s->next=node_s->prev=node_s;
else{after=child->next;child ->next=node_s;node_s->prev=child;node_s->next=after;after->prev=node_s;}}node_c=node_r;
fh.deg_pointer[dg]=NNULL;}if(node_l==last) break;node_c=node_n;}fh.dist=VERY_FAR;for(dg=0;dg<fh.deg_max;dg ++){if(
fh.deg_pointer[dg] != NNULL){node_r=fh.deg_pointer[dg];fh.deg_pointer[dg]=NNULL;Check_min(node_r);node_r->heap_parent=NNULL;
}}}else fh.min=NNULL;}return nd;}int dikf(n,nodes,source) long n;node *nodes,*source;{long dist_new,dist_old,dist_from;
long pos_new,pos_old;node *node_from,*node_to,*node_last,*i;arc *arc_ij,*arc_last;long num_scans=0;Init_fheap(n);node_last=
nodes+n ;for(i=nodes;i != node_last;i++){i->parent=NNULL;i->dist=VERY_FAR;}source->parent=source;source->dist=0;
Insert_to_fheap(source);while(1){node_from=Extract_min();if(node_from==NNULL)break;num_scans ++;arc_last =(node_from+1)
->first;dist_from=node_from->dist;for(arc_ij=node_from->first;arc_ij != arc_last;arc_ij ++){node_to =arc_ij->head;
dist_new=dist_from+(arc_ij->len);if(dist_new<node_to->dist){node_to->dist=dist_new;node_to->parent=node_from;if(
NODE_IN_FHEAP(node_to)) {Fheap_decrease_key(node_to);} else {Insert_to_fheap(node_to);} }}}n_scans=num_scans;return (0);}
```

Comparison: CHR implementation


```
:- use_module(library(chr)).
:- constraints edge(+dense_int,+,+), distance(+dense_int,+),
               dijkstra(+), scan(+,+), relabel(+,+),
               extract_min(?,?), decr_or_ins(+,+), mark(+),
               ch2rt(+), decr(+,+,+,+,+mark), min(+,+),
               item(+dense_int,+,+,+dense_int,+), findmin.

dijkstra(A) <=> scan(A,0).
scan(N,L) ==> distance(N,L).
scan(N,L), edge(N,N2,W) ==> L2 is L+W, relabel(N2,L2).
scan(N,L)<=> extract_min(N2,L2) | scan(N2,L2).
scan(N,L) <=> true.
distance(N,_) \ relabel(N,_) <=> true.
relabel(N,L) <=> decr_or_ins(N,L).

min(_,A) \ min(_,B) <=> A =< B | true.
extract_min(X,Y), min(I,K), item(I,_,_,_,_)
        <=> ch2rt(I), findmin, X=I, Y=K.
extract_min(_,_) <=> fail.
ch2rt(I) \ item(C,K,R,I,_) <=> item(C,K,R,0,u).
ch2rt(I) <=> true.
findmin, item(I,K,_,0,_) ==> min(I,K).
findmin <=> true.

item(I1,K1,R,0,_), item(I2,K2,R,0,_) <=> K1 < K2 |
        R1 is R+1, item(I2,K2,R,I1,u), item(I1,K1,R1,0,u).

item(I,O,R,P,M), decr_or_ins(I,K) <=> K < O | decr(I,K,R,P,M).
item(I,O,_,_,_) \ decr_or_ins(I,K) <=> K >= O | true.
decr_or_ins(I,K) <=> item(I,K,0,0,u), min(I,K).

decr(I,K,_,_,_) ==> min(I,K).
decr(I,K,R,0,_) <=> item(I,K,R,0,u).
item(P,PK,_,_,_) \ decr(I,K,R,P,M) <=> K >= PK | item(I,K,R,P,M).
decr(I,K,R,P,M) <=> item(I,K,R,0,u), mark(P).
mark(I), item(I,K,R,0,_) <=> R1 is R-1, item(I,K,R1,0,u).
mark(I), item(I,K,R,P,m) <=> R1 is R-1, item(I,K,R1,0,u), mark(P).
mark(I), item(I,K,R,P,u) <=> R1 is R-1, item(I,K,R1,P,m).
```

Overview

Complexity

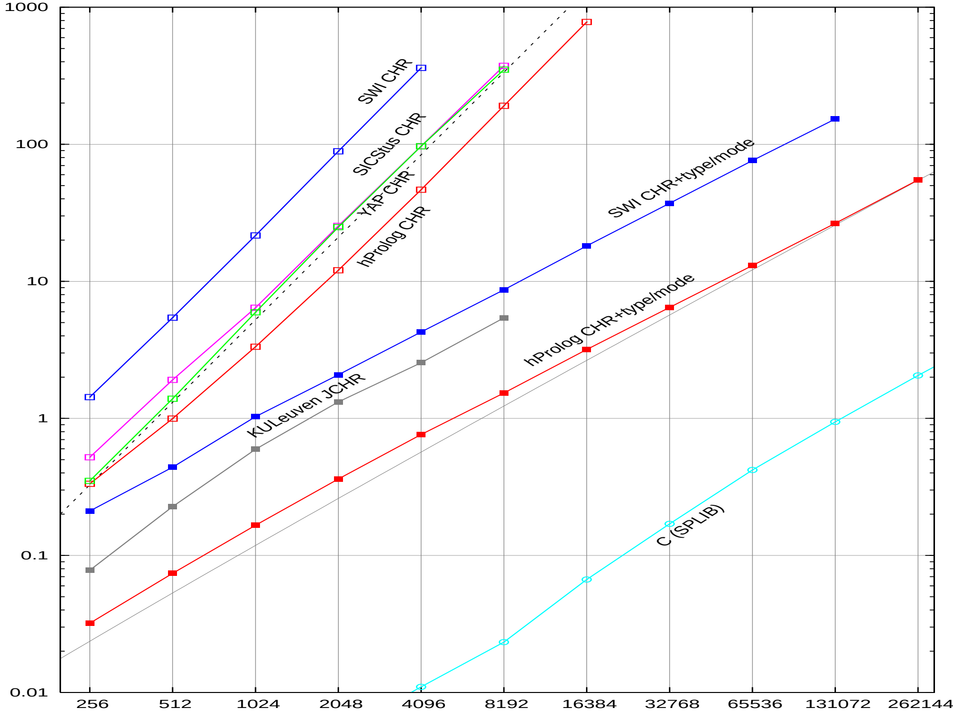
- Dijkstra takes $O(nI + mD + nE)$ time
  where I, D, E is the time for insert, decrease-key, extract-min

- Fibonacci heap: $I = D = O(1)$ (amortized)

- Extract-min: $O(\log n)$ (amortized)
  - Reason: a node with rank k has at least F_{k+2} descendants (F_i is the i-th Fibonacci number)
  - Hence the maximal rank is $O(\log n)$
  - So `extract_min` adds $O(\log n)$ children
    and `findmin` looks at $O(\log n)$ roots

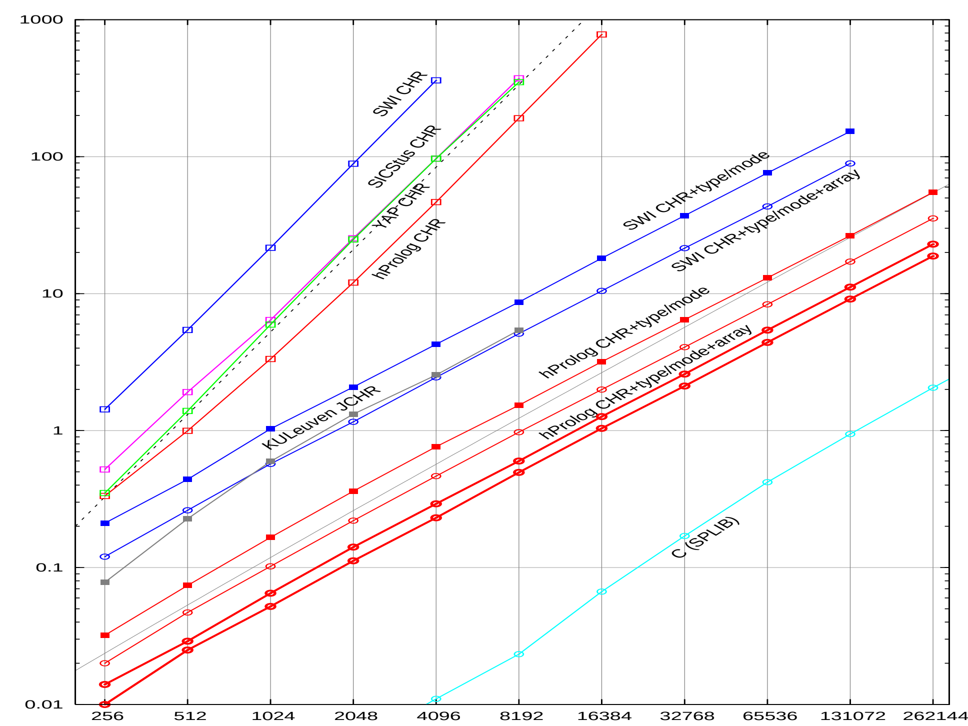
Optimal complexity in CHR?

- To get the optimal complexity, the constraint store operations have to be fast enough
- Adding mode declarations suffices
  (this allows the compiler to use hashtables with $O(1)$ insert/remove/lookup)
- Experimental setup: “Rand-4” (sparse graphs)


1000
100
10
1
0.1
0.01
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
SWI CHR
SICStus CHR
YAP CHR
hProlog CHR
SWI CHR+type/mode
hProlog CHR+type/mode
KULeuven JCHR
C (SPLIB)

Constant factors

- What about the constant factors?
- To improve constant factors: **array** constraint store instead of hashtable store
- New built-in type `dense_int` for ground arguments in $[0, n]$, array store used to index on such arguments
- For this program: 35% to 40% faster than hashtables



1000
100
10
1
0.1
0.01
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
SWI CHR
SICStus CHR
YAP CHR
hProlog CHR
SWI CHR+type/mode
SWI CHR+type/mode+array
KULeuven JCHR
hProlog CHR+type/mode
hProlog CHR+type/mode+array
C (SPLIB)

Experimental results

- Optimal complexity is achieved in practice
- Constant factors:
  about 10 times slower than C implementation

$$\frac{CHR}{C} \approx 10$$

Overview

Conclusion

- Readable, compact, executable and reasonably efficient CHR description of Dijkstra's algorithm with Fibonacci heaps
- Probably first implementation of Fibonacci heaps in a declarative language
  - [King 1994] has functional binomial queues, which is simpler but asymptotically slower (about 45 lines of Haskell)
  - [Okasaki 1996], [Brodal 1996] have many priority queues but not Fibonacci heaps
  - Probably no natural functional encoding of F-heaps
  - [McAllester 1999] has very compact logical rules for Dijkstra's algorithm which takes $O(m \log m)$ time, but this takes an interpreter with built-in F-heaps

Future work

- Challenge: improve the constant factor until

$$\frac{CHR}{C} < k$$

(what k can we wish for? $k = 5$? $k = 2$? why not $k = 1$?)

- CHR for host-language C ? (maybe based on Java CHR)
- High-level algorithm descriptions in CHR
  - "executable pseudocode"
  - with only marginal performance penalty

Appendix

Fibonacci Heap operations

- Insert is easy: add new root pair and candidate minimum

```
insert(I,K) <=> item(I,K,0,0,u), min(I,K).
```

- Extract minimum: remove, children2roots, find new minimum

```
extract_min(X,Y), min(I,K), item(I,_,_,_,_)
  <=> ch2rt(I), findmin, X=I, Y=K.
extract_min(_,_) <=> fail.
```

- Children2roots:

```
ch2rt(I) \ item(C,K,R,I,_) <=> item(C,K,R,0,u).
ch2rt(I) <=> true.
```

- Find new minimum: only search roots!

```
findmin, item(I,K,_,0,_) ==> min(I,K).
findmin <=> true.
```

Decrease-key-or-insert

- New key smaller: decrease key

```
item(I,O,R,P,M), decr_or_ins(I,K)
  <=> K<O | decr(I,K,R,P,M).
```

(note: `item/5` is removed, `decr/5` will re-insert it)

- New key bigger: do nothing

```
item(I,O,_,_,_) \ decr_or_ins(I,K)
  <=> K >= O | true.
```

- No such item in the queue: insert

```
decr_or_ins(I,K) <=> insert(I,K).
```

Decrease-key

- Maybe new minimum:
```
decr(I,K,_,_,_) ==> min(I,K).
```
- Decreasing the key of a root is easy
```
decr(I,K,R,0,_) <=> item(I,K,R,0,u).
```
- If the new key is still larger than the parent key, no problem:
```
item(P,PK,_,_,_) \ decr(I,K,R,P,M)
  <=> K>=PK | item(I,K,R,P,M).
```
- Otherwise, make the pair a new root (*cut*) and mark its parent
```
decr(I,K,R,P,M) <=> item(I,K,R,0,u), mark(P).
```

Marking a node

- Lose one child: ok. Lose two: not ok → *cascading cut*
- Node is marked if it has lost a child
- Roots are always unmarked (`u`):
```
mark(I), item(I,K,R,0,_) <=> item(I,K,R-1,0,u).
```
- Unmarked node becomes marked (`m`):
```
mark(I), item(I,K,R,P,u) <=> item(I,K,R-1,P,m).
```
- Already marked node is cut and its parent is marked:
```
mark(I), item(I,K,R,P,m)
  <=> item(I,K,R-1,0,u), mark(P).
```