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
Overview

1 Introduction

2 Single-source shortest path
   - Problem
   - Dijkstra’s Algorithm
   - Priority queues

3 Fibonacci Heaps

4 Performance
   - Complexity
   - Benchmarking

5 Conclusion
   - Conclusion
   - Future work
1 Introduction

2 Single-source shortest path
   • Problem
   • Dijkstra’s Algorithm
   • Priority queues

3 Fibonacci Heaps

4 Performance
   • Complexity
   • Benchmarking

5 Conclusion
   • Conclusion
   • Future work
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]
Can all algorithms be implemented in a natural, elegant, compact way?

Some empirical evidence
e.g. union-find [Schrijvers-Fruhwirth TPLP 2006]

and: What about constant factors?
Overview

1. Introduction

2. Single-source shortest path
   - Problem
   - Dijkstra’s Algorithm
   - Priority queues

3. Fibonacci Heaps

4. Performance
   - Complexity
   - Benchmarking

5. Conclusion
   - Conclusion
   - Future work
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
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”
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 $\text{distance}(X,\_)$ constraint

We start by scanning the source:
\[ \text{dijkstra}(A) \iff \text{scan}(A,0). \]
Dijkstra's Algorithm

Jon Sneyers, Tom Schrijvers, Bart Demoen

Dijkstra's Algorithm and Fibonacci Heaps in CHR
Scanning a node

- Scanning a node: first make it scanned
  \[ \text{scan}(N,L) \Rightarrow \text{distance}(N,L). \]

- Then label its neighbours:
  \[ \text{scan}(N,L), \text{edge}(N,N2,W) \Rightarrow \text{relabel}(N2,L+W). \]

- Finally, pick the next node to scan. Pick a labeled node with the smallest label:
  \[ \text{scan}(N,L) \Leftrightarrow \text{extract min}(N2,L2) \mid \text{scan}(N2,L2). \]

- If there is no next node, stop:
  \[ \text{scan}(N,L) \Leftrightarrow \text{true}. \]
(re)labeling a node: do nothing if it is already scanned
\[ \text{distance}(N,\_ ) \setminus \text{relabel}(N,\_ ) \leftrightarrow \text{true}. \]

Otherwise, add or decrease its label:
\[ \text{relabel}(N,L) \leftrightarrow \text{decr_or_ins}(N,L). \]

Still need to define \text{decr_or_ins}/2 and \text{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: \texttt{extract\_min}/2 in $O(1)$, \texttt{decr\_or\_ins}/2 in $O(n)$
  \quad $\rightarrow$ Dijkstra in $O(mn)$ ($m$ edges, $n$ nodes)

- Array: \texttt{extract\_min}/2 in $O(n)$, \texttt{decr\_or\_ins}/2 in $O(1)$
  \quad $\rightarrow$ Dijkstra in $O(n^2)$

- Binary heap: \texttt{extract\_min}/2 and \texttt{decr\_or\_ins}/2 in $O(\log n)$
  \quad $\rightarrow$ Dijkstra in $O(m \log n)$
Overview

1 Introduction

2 Single-source shortest path
   - Problem
   - Dijkstra’s Algorithm
   - Priority queues

3 Fibonacci Heaps

4 Performance
   - Complexity
   - Benchmarking

5 Conclusion
   - Conclusion
   - Future work
Fibonacci Heaps [Fredman-Tarjan 1987]

- Advanced priority queue
- \( \text{extract\_min/2 in } O(\log n), \text{decr\_or\_ins/2 in } O(1) \)
  \( \rightarrow \) Dijkstra in \( O(m + n \log n) \)
- Optimal for Dijkstra-based shortest path!
Store the pairs as item/5 constraints:

\[ \text{item(Item,Key,Rank,Parent,Mark)} \]

- Parent is 0 if the pair is a root, > 0 otherwise
- \( \text{Rank} = \text{number of children} \)
Maintain the current minimal pair:
\[ \min(_,A) \setminus \min(_,B) \iff A \leq B \mid \text{true.} \]

Heap-ordered trees: parent has smaller key than children
\[ \to \] minimum must be a root

No two roots can have the same rank:
\[ \text{item}(A,K1,R,0,_) , \text{item}(B,K2,R,0,_) \iff K1 \leq K2 \mid \text{item}(A,K1,R+1,0,u), \text{item}(B,K2,R,I1,u). \]
Fibonacci Heap operations

- Insert is easy: add new root pair and candidate minimum
  \[ \text{insert}(I,K) \iff \text{item}(I,K,0,0,0,u), \text{min}(I,K). \]

- Extract minimum: remove, children2roots, find new minimum
  \[ \text{extract}\_\text{min}(X,Y), \text{min}(I,K), \text{item}(I,\_\_\_,\_\_\_), \]
  \[ \iff \text{ch2rt}(I), \text{findmin}, X=I, Y=K. \]
  \[ \text{extract}\_\text{min}(\_\_,\_) \iff \text{fail}. \]

- Children2roots:
  \[ \text{ch2rt}(I) \setminus \text{item}(C,K,R,I,\_) \iff \text{item}(C,K,R,0,u). \]
  \[ \text{ch2rt}(I) \iff \text{true}. \]

- Find new minimum: only search roots!
  \[ \text{findmin}, \text{item}(I,K,\_\_,\_\_,\_\_) \implies \text{min}(I,K). \]
  \[ \text{findmin} \iff \text{true}. \]
Decrease-key-or-insert

- New key smaller: decrease key
  \[ \text{item}(I,0,R,P,M), \text{decr}_\text{or}_\text{ins}(I,K) \]
  \[ \iff \ K < 0 \quad \text{or} \quad \text{decr}(I,K,R,P,M). \]
  (Note: \text{item}/5 is removed, \text{decr}/5 will re-insert it)

- New key bigger: do nothing
  \[ \text{item}(I,0,_,_,_,_) \setminus \text{decr}_\text{or}_\text{ins}(I,K) \]
  \[ \iff \ K \geq 0 \quad \text{or} \quad \text{true}. \]

- No such item in the queue: insert
  \[ \text{decr}_\text{or}_\text{ins}(I,K) \iff \text{insert}(I,K). \]
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?
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) (node->nodes + 1) #define MARKED 2 #define IN_HEAP 1 #define NNULL (node*) NULL #define NOT_ENOUGH_MEM 2
define 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,*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->min=nh=dist=0;}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)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(!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]=NNULL;}else{if(node_c->dist<node_r->dist){node_s=node_r;node_r=node_c;} else node_s=node_c;}}if(node_s==node_c)first=child->next;else{after=child->next;child->prev=first;first->next=child;child->prev=first;before->next=after;after->prev=before;}}}else{fh.min=NNULL;}}} Dijkstra's Algorithm and Fibonacci Heaps in CHR
Comparison: CHR implementation

Dijkstra's Algorithm and Fibonacci Heaps in CHR

int dikf(n, nodes, source)
long n; node *nodes, *source;
{
long dist_new, dist_old, dist_from;
node *node_from, *node_to, *node_last;
arc *arc_ij, *arc_last;
long num_scans = 0;

Init_fheap(n);
node_last = NULL;

for (i = 0; i != nodes; i++)
{
node_c = i;
node_c->parent = NNULL;
node_c->dist = VERY_FAR;
source->parent = source;
source->dist = 0;
}
Insert_to_fheap(source);

while (1)
{
node_from = Extract_min();
if (node_from == NULL) break;
num_scans ++;
arc_last = (node_from + 1);
node_s = new node;
node_s->status = IN_HEAP;
node_s->son = node_r->son;
node_s->next = node_r->next;
node_r->next = node_s;
node_s->prev = node_r;
node_r->deg ++;
node_s->dist = 0;
mark(node_s);
for (arc_ij = node_s->first; arc_ij != arc_last; arc_ij++)
{
node_to = arc_ij->head;
if (NODE_IN_FHEAP(node_to)) {Fheap_decrease_key(node_to);}
else {Insert_to_fheap(node_to);}
}
}

return (0);
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
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)
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
Runtime (excluding garbage collection) in seconds (logscale)

Nodes (logscale) \( (m=4n) \)

SWI CHR
SICStus CHR
YAP CHR
hProlog CHR

KULeuven JCHR
hProlog CHR+type/mode
hProlog CHR+type/mode+array

C (SPLIB)

\( O(n^2) \)
\( O(n \log n) \)
Optimal complexity is achieved in practice

Constant factors:
about 10 times slower than C implementation

\[
\frac{CHR}{C} \approx 10
\]
Overview

1. Introduction
2. Single-source shortest path
   - Problem
   - Dijkstra’s Algorithm
   - Priority queues
3. Fibonacci Heaps
4. Performance
   - Complexity
   - Benchmarking
5. Conclusion
   - Conclusion
   - Future work
- 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 binominal 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

6 Fibonacci Heap operations
Fibonacci Heap operations

- Insert is easy: add new root pair and candidate minimum
  \[
  \text{insert}(I,K) \leftrightarrow \text{item}(I,K,0,0,u), \text{min}(I,K).
  \]

- Extract minimum: remove, children2roots, find new minimum
  \[
  \text{extract}\_\text{min}(X,Y), \text{min}(I,K), \text{item}(I,_,_,_,_)
  \leftrightarrow \text{ch2rt}(I), \text{findmin}, X=I, Y=K.
  \]

  \[
  \text{extract}\_\text{min}(_,_) \leftrightarrow \text{fail}.
  \]

- Children2roots:
  \[
  \text{ch2rt}(I) \setminus \text{item}(C,K,R,I,_) \leftrightarrow \text{item}(C,K,R,0,u).
  \]

  \[
  \text{ch2rt}(I) \leftrightarrow \text{true}.
  \]

- Find new minimum: only search roots!
  \[
  \text{findmin}, \text{item}(I,K,_,0,_) \Rightarrow \text{min}(I,K).
  \]

  \[
  \text{findmin} \leftrightarrow \text{true}.
  \]
Fibonacci Heap operations

Decrease-key-or-insert

- New key smaller: decrease key
  \[ \text{item}(I,0,R,P,M), \ \text{decr}\_or\_ins(I,K) \]
  \[ \iff K < 0 \lor \text{decr}(I,K,R,P,M). \]
  (note: \text{item}/5 is removed, \text{decr}/5 will re-insert it)

- New key bigger: do nothing
  \[ \text{item}(I,0,_,_,_,_) \setminus \text{decr}\_or\_ins(I,K) \]
  \[ \iff K \geq 0 \lor \text{true}. \]

- No such item in the queue: insert
  \[ \text{decr}\_or\_ins(I,K) \iff \text{insert}(I,K). \]
Fibonacci Heap operations

Decrease-key

- Maybe new minimum:
  \( \text{decr}(I,K,_,_,_) \Rightarrow \text{min}(I,K) \).

- Decreasing the key of a root is easy
  \( \text{decr}(I,K,R,0,_) \Leftrightarrow \text{item}(I,K,R,0,u) \).

- If the new key is still larger than the parent key, no problem:
  \( \text{item}(P,PK,_,_,_) \setminus \text{decr}(I,K,R,P,M) \)
  \( \Leftrightarrow K=PK \lor \text{item}(I,K,R,P,M) \).

- Otherwise, make the pair a new root (cut) and mark its parent
  \( \text{decr}(I,K,R,P,M) \Leftrightarrow \text{item}(I,K,R,0,u), \text{mark}(P) \).
Fibonacci Heap operations

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):
  \[ \text{mark}(I), \text{item}(I,K,R,0,u) \Leftrightarrow \text{item}(I,K,R-1,0,u). \]
- Unmarked node becomes marked (m):
  \[ \text{mark}(I), \text{item}(I,K,R,P,u) \Leftrightarrow \text{item}(I,K,R-1,P,m). \]
- Already marked node is cut and its parent is marked:
  \[ \text{mark}(I), \text{item}(I,K,R,P,m) \Leftrightarrow \text{item}(I,K,R-1,0,u), \text{mark}(P). \]