Dynamic Shortest Path Algorithms

Professor Dr. Petra Mutzel

Lehrstuhl für Algorithm Engineering

Fakultät für Informatik

TU Dortmund

Selected Topics on Combinatorial Optimization

Vienna Graduate School on Computational Optimization

Lecture 7, November 13, 2018

Outline

Literature

- G. Ramalingam and T. Reps: On the computational complexity of dynamic graph problems. Theoretical Computer Science 158, 1996, 233-277
- C. Demetrescu and G. F. Italiano: Experimental Analysis of Dynamic All Pairs Shortest Path Algorithms, ACM Transactions on Algorithms, vol. 2 (4), 2006, S. 578-601
- C. Demetrescu and G. F. Italiano: A new approach to dynamic all pairs shortest paths. Journal of the ACM 51(6): 968–992 (2004)

There are many recent papers for dynamic algorithms for solving the All-Pairs Shortest Path Problem (social network analysis); they are mainly based on Ramalingam and Reps.

Dynamic Graph Algorithms

Definition (Dynamic Graph Algorithm)

A dynamic algorithm maintains a property P of a weighted graph during dynamic changes of, e.g.,

- insertion of new edges/vertices
- deletion of new edges/vertices
- weight-changes of edges/vertices

Requirement to dynamic algorithms

- fast response of queries concerning property P
- fast update operations, in particularly faster than a static algorithm computing from scratch

Here, we are concerned with edge changes, only.

Motivation

Applications

- Prediction of future states of networks shall be answered quickly
- Centrality indices for social network analysis are based on shortest path
- Social networks change (quickly) over time
- Brain networks change very quickly
- Weights on road networks vary a lot during day time
- Navigation queries shall be answered quickly

Dynamic Complexity Measures

Amortized

Worst Case in the size of the input, amortized over a sequence of operations.

Alternatively

Worst Case in the size of the changes in input and output.

Ramalingam and Reps used the alternative complexity measure.

Dynamic Complexity Measures

Definition

- A vertex v is called modified if the input value has changed or if v was inserted or deleted.
- A vertex v is called affected if v was inserted or it got a new output value.
- Let *CHANGED* be the set of all modified or affected vertices.
- We define $\delta = |CHANGED|$.
- We define $\epsilon = \delta +$number of edges incident to *CHANGED*.

Remarks

- δ is a lower bound for the number of operations needed.
- It is also obvious that all the incident edges to *CHANGED* need to be tested in order to guarantee correctness.

Dynamic Complexity Measures

Definition

- A dynamic algorithm is called full-dynamic if it works for increased edge weights and edge deletions, resp., as well as for edge weight decrease and edge insertions, resp.
- Otherwise it is called incremental (or partially dynamic).

Definition

- A dynamic algorithm is called bounded if the update time for one operation is bounded in ϵ.
- Otherwise it is called unbounded.

Here: a bounded dynamic single-source shortest path algorithm.

Single-Source Shortest Path

Definition (SSSP)

- **Given:** Directed graph $G = (V, E)$ with positive edge weights c and vertex $t \in V$.
- **Find:** Shortest (v, t)-paths in G for all $v \in V$.

Remark: Our definition is different from the standard definition, where a path form s to all other vertices is searched.

Dijkstra's Algorithm

Let M be a very large number and $dist(v)$ the distance matrix.

Q is a priority queue with operations

- InsertQ($v, dist(v)$): inserts v into Q with priority $dist(v)$
- ExtractMinQ(): outputs and then deletes the minimum from Q
- DecreasePrioQ($v, dist(v)$): updates the decreased priority of v in Q to $dist(v)$

Dijkstra's Algorithm

```
1:  function DIJKSTRA
2:      Initialisation: Q = ∅; dist[t] = 0; INSERTQ(t, 0)
3:      for all vertices v ∈ V, v ≠ t do dist(v) = M; INSERTQ(v, M)
4:      while Q ≠ ∅ do
5:          v =EXTRACTMINQ()          // distance of v is minimum
6:          for all edges (u, v) ∈ E  do // edge scanning
7:              if dist(u) > c(u, v) + dist(v) then
8:                  set dist(u) = c(u, v) + dist(v)
9:                  DECREASEPRIOQ(u, dist(u))
10:             end if
11:         end for
12:     end while
13:     return dist(v) for all v ∈ V
14: end function
```

Analysis of Dijkstra's Algorithm

Let $n = |V|$ and $m = |E|$.

The running time of Dijkstra is $T(n, m) =$
$n \cdot T(\textsc{InsertQ}) + n \cdot T(\textsc{ExtractMinQ}() + m \cdot T(\textsc{DecreasePrioQ}())$

Binary Heap for Q: $T(n, m) = O((n + m) \log n)$

- $T(\textsc{InsertQ}) = O(\log n)$
- $T(\textsc{ExtractMinQ}() = O(\log n)$
- $T(\textsc{DecreasePrioQ}() = O(\log n)$

Fibonacci Heap for Q: $T(n, m) = O(m + n \log n)$

- $T(\textsc{InsertQ}) = O(1)$
- $T(\textsc{ExtractMinQ}() = O(\log n)$ amortized
- $T(\textsc{DecreasePrioQ}() = O(1)$ amortized

Algorithm by Ramalingam and Reps

Algorithm supports the operations

- DelEdge(v, w): delete the edge (v, w) from $G \rightarrow$ we get G'
- InsEdge(v, w, c): insert the edge (v, w) with weight c into $G \rightarrow$ we get G'

Remark: Using the operations above, also the operations IncreaseWeight(v, w, c) and DecreaseWeight(v, w, c) can be simulated.

Algorithm supports the queries

- $dist(v)$: output distance between vertices v and t
- $path(v)$: output a shortest path from v to t if there exists one

Basics of the Algorithm

Definition

A subgraph T of G is called shortest-path tree for G with sink t if

- T is a directed tree with root t
- $V(T)$ is a set of vertices that can reach t, and
- for each edge (u, v) in T we have: $dist(u) = dist(v) + c(u, v)$.

Basics of the Algorithm

Definition

- An edge is called SP-edge, if it is part of the shortest (v, t)-path for some $v \in V$.
- Hence, an edge is an SP-edge iff $dist(u) = dist(v) + c(u, v)$.
- Let $SP(G)$ be the subgraph of G induced by the set of all SP-edges.

Observation

- Every shortest path is part of $SP(G)$ and vice versa:
- Every path in $SP(G)$ is a shortest path in G.
- Since all edge weights are positive, $SP(G)$ is a directed acyclic graph.

Operation DeleteEdge(v, w)

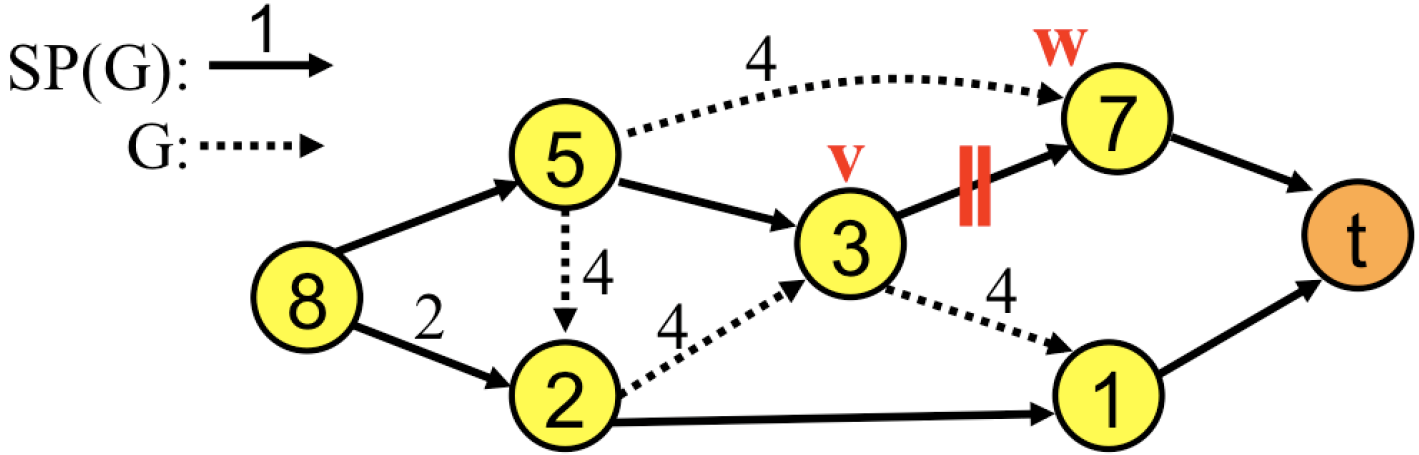
DeleteEdge(v, w) from $G \to G'$

Remark

- A vertex v is affected if v was inserted or it got a new output value.
- Hence: a vertex v in G' is affected if $dist_G(v) \neq dist_{G'}(v)$.

Definition

An SP-edge (x, y) is called affected by DeleteEdge(v, w) if there is no (x, t)-path in G' with length $dist_G(x)$ using the edge (x, y).



Operation DeleteEdge(v, w)

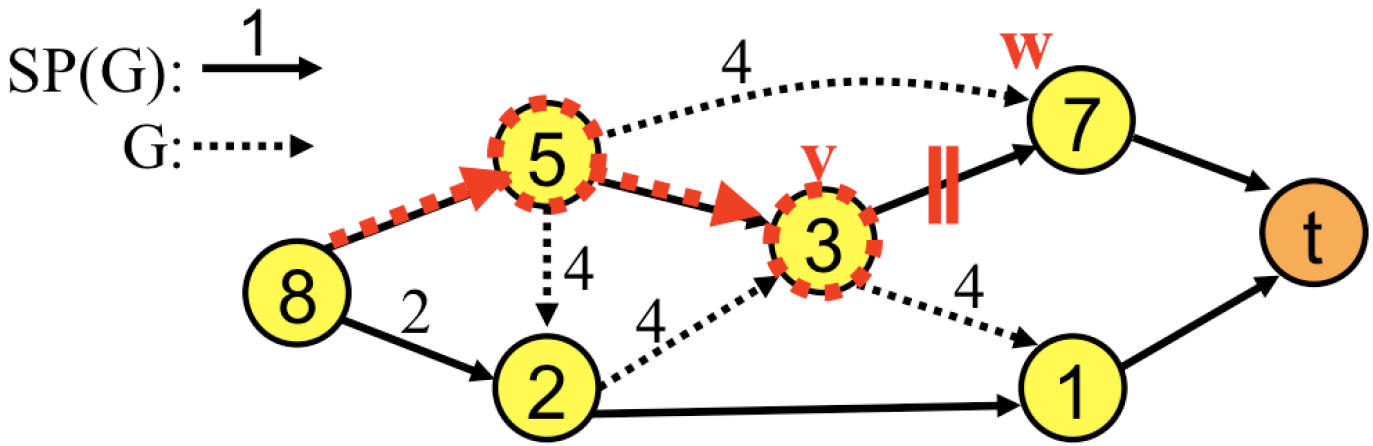
DeleteEdge(v, w) from $G \to G'$

Remark

- A vertex v is affected if v was inserted or it got a new output value.
- Hence: a vertex v in G' is affected if $dist_G(v) \neq dist_{G'}(v)$.

Definition

An SP-edge (x, y) is called affected by DeleteEdge(v, w) if there is no (x, t)-path in G' with length $dist_G(x)$ using the edge (x, y).



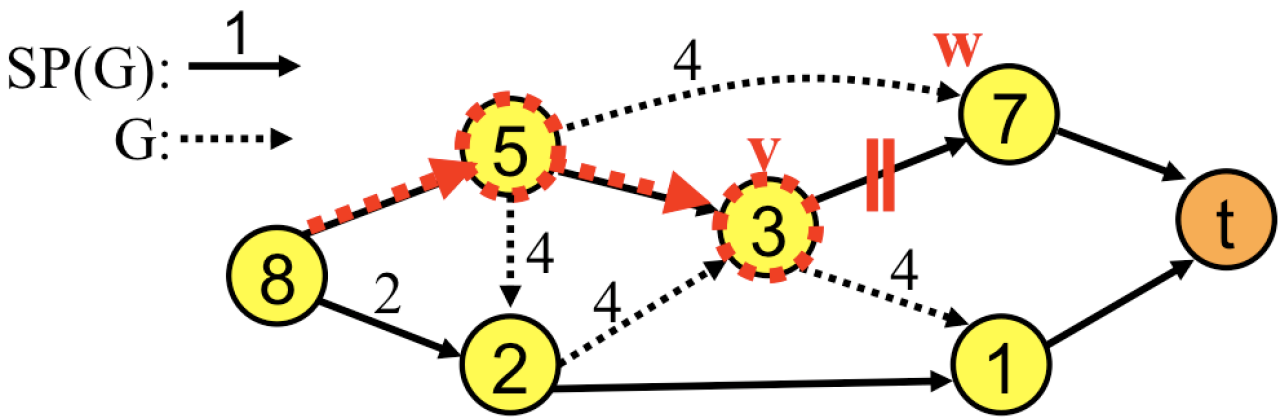
Operation DeleteEdge(v, w)

Definition

An SP-edge (x, y) is called affected by $\textsc{DeleteEdge}(v, w)$ if there is no (x, t)-path in G' with length $dist_G(x)$ using the edge (x, y).

Observations in $SP(G)$

- (x, y) is affected $\Leftrightarrow$ y is affected
- vertex $x \neq v$ is affected $\Leftrightarrow$ all outgoing edges of x are affected
- v itself is affected $\Leftrightarrow$ (v, w) is the only outgoing SP-edge from v



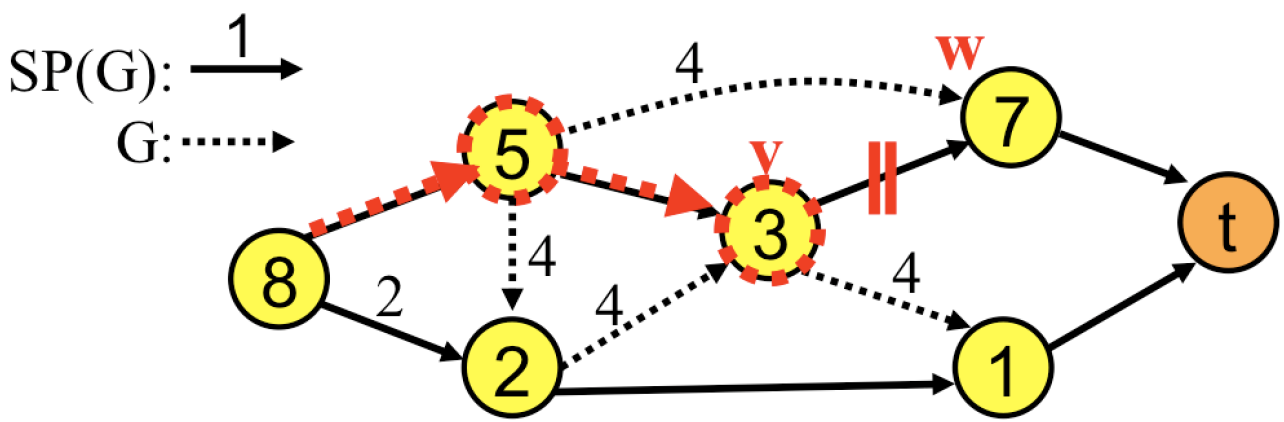
Algorithmic Idea for DeleteEdge(v, w)

Phase 1:

- Find the set of all affected edges and vertices
- Delete the affected edges from $SP(G)$

Phase 2:

- Compute the new $dist()$ values for the affected vertices
- Update $SP(G)$



Algorithmic Idea for Phase 1 of DeleteEdge(v, w)

Observation

The set of affected vertices corresponds to the set of vertices u for which there is no (u, t)-path in $SP(G) \setminus (v, w)$.

Algorithmic Idea of Phase 1:

- Let $H = SP(G) \setminus (v, w)$.
- While there exists a vertex u in H with $outdeg(v) = 0$:
  - Delete u from H and add u to the set of affected vertices.

Remarks

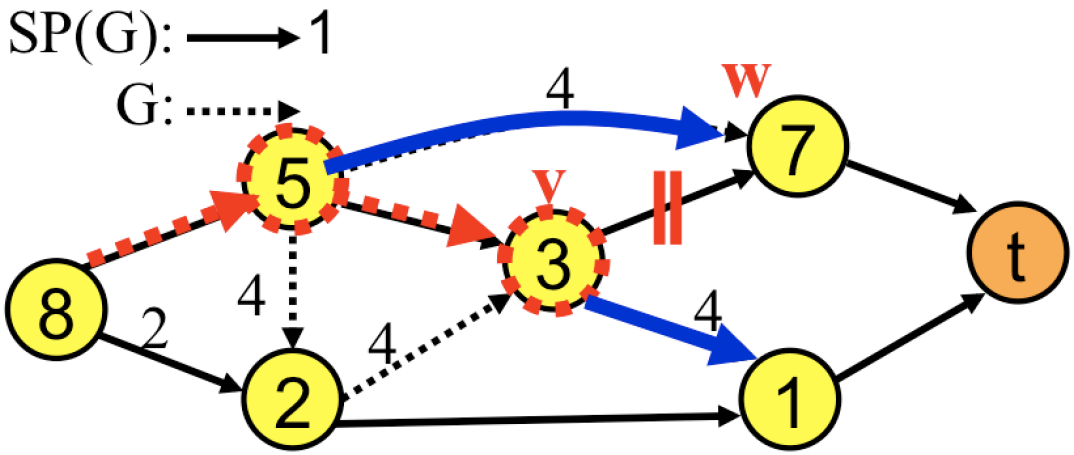
- The algorithm realizes Topological Sort
- Please note that BFS is not working!
- $SP(G)$ is maintained either as graph with adjacency lists or implicitly (test if $dist(u) = dist(v) + c(u, v)$.

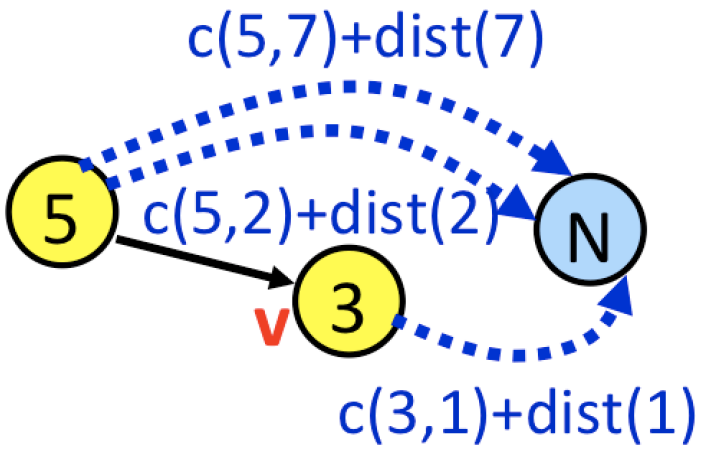
Algorithmic Idea for Phase 2 of DeleteEdge(v, w)

- Let N be the set of non-affected vertices.
- **Objective:** Compute $dist(x)$ for all affected vertices in $A = V \setminus N$.

Algorithmic Idea of Phase 2:

- Let $H = SP(G) \setminus (v, w)$.
- Think of contracting N to a vertex t' and substitute the edges (x, y) with $x \in A$, $y \in N$ by (x, t') with weights $x(x, y) + dist(y)$
- Run the Dijkstra algorithm on the new graph





Algorithm for Phase 2 of DeleteEdge(v, w)

Algorithm for Phase 2:

- We do not really contract N, since this would need too much time

Initialization of Dijkstra

- For all affected vertices a: calculate
  $dist(a) = \min(\{c(a,b) + dist(b) \mid (a,b) \in E \text{ and } b \text{ not affected } \}, M)$
- If $dist(a) \neq M$ then $\textsc{InsertQ}(a, dist(a))$

Main Loop of Dijkstra after $a \leftarrow$ ExtractMinQ():

- For each $b \in succ(a)$: If $dist(a) == c(a,b) + dist(b)$, then insert (a,b) to $SP(G)$
- For each $b \in pred(a)$: If $dist(b) > c(b,a) + dist(a)$, then $dist(b) = c(b,a) + dist(a)$; Only then call:
- $\textsc{DecreasePrioQ}(b, dist(b))$ resp. $\textsc{InsertQ}(b, dist(b))$

Algorithm Analysis of DeleteEdge(v, w)

Correctness of Algorithm for Phase 2:

- Obviously, affected vertex set will be computed correctly
- All affected vertices that are able to reach t will be inserted into Q during the Dijkstra run.
- Their distance will be set correctly due to the correctness of Dijkstra
- Non-affected vertices will never be inserted into Q
- The SP-graph will be maintained correctly.

Algorithm Analysis of DeleteEdge(v, w)

Running time for Phase 1:

- Number of iterations: δ
- One iteration for vertex u needs $O(|pred(u)|)$
- Total running time: $O(\epsilon)$

Running time for Phase 2 (Fibonacci Heaps):

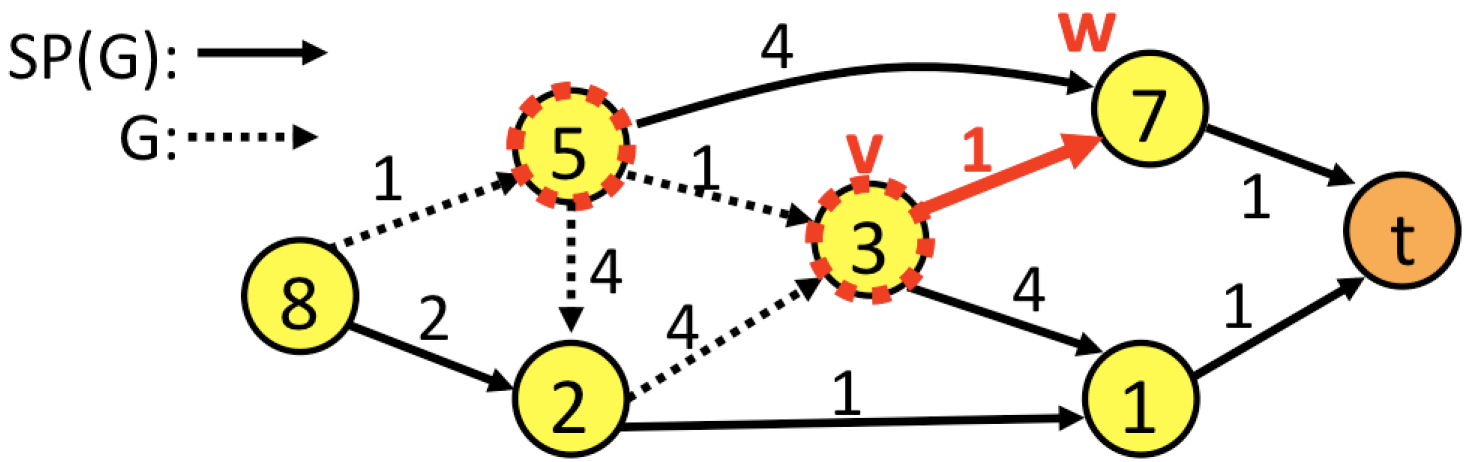
- $T(\text{INSERTQ}) = O(1)$
- $T(\text{DECREASEPRIOQ}() = O(1)$ amortized
- $T(\text{EXTRACTMINQ}() = O(\log p)$ amortized with p is the number of elements in the heap
- Number of iterations: at most δ
- Total running time: $O(\epsilon + \delta \log \delta)$

Operation InsertEdge(v, w)

InsertEdge(v, w) from $G \to G'$

Remark

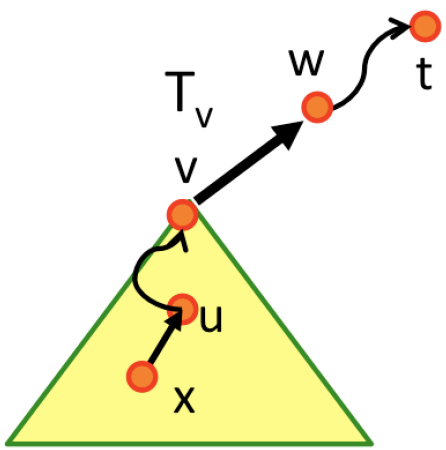
- If a vertex u is affected then all the new shortest (u, t)-paths in G' are of the form:
- (shortest (u, v)-path, edge (v, w), shortest (w, t)-path)
- u is affected $\Leftrightarrow dist_{G'}(u, v) + c(v, w) + dist_G(w) < dist_G(u)$, where $dist_{G'}(u, v)$ is the length of the shortest path from u to v in G', which is equal to $dist_G(u, v)$ for this case.



Operation InsertEdge(v, w)

Consider the new shortest path tree T_v for vertex v:

- Let $x \neq v$ be an arbitrary vertex and (x, u) an edge in T_v.
- If x is affected, then also u must be affected, because otherwise:
- If there is a shortest (u, t)-path P not using the edge (v, w), then the path $((x, u), P)$ would also be a shortest (x, t) path and x would not be affected.
- $\Rightarrow$ The set of affected vertices builds a connected subtree of T_v with root v



Algorithm for InsertEdge(v, w)

Algorithmic Idea for InsertEdge(v, w)

- Use Dijkstra with priority $dist(x) - dist(v)$ for all affected vertices x
- Since $dist(v)$ is a constant for all vertices during the run, we can instead calculate with the distances $dist(x)$

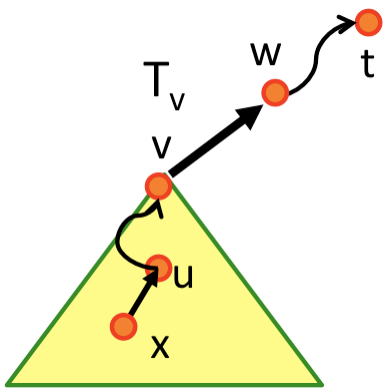
Initialization of Dijkstra

- If v is affected, then $\textsc{InsertQ}(v, dist_{G'}(v))$
- Otherwise: If there exists a shortest path in G' using the edge (v, w) then update $SP(G)$.

Algorithm for InsertEdge(v, w)

Main Loop of Dijkstra after $u \leftarrow$ ExtractMinQ():

- Delete all outgoing edges from u in $SP(G)$
- For each $y \in succ(u)$: If $dist(u) == c(u, y) + dist(y)$, then insert (u, y) to $SP(G)$
- For each $x \in pred(u)$: If $dist(x) > c(x, u) + dist(u)$, then $dist(x) = c(x, u) + dist(u)$; Only then call:
- DecreasePrioQ$(x, dist(x))$ resp. InsertQ$(x, dist(x))$
- Otherwise If $dist(x) == c(x, u) + dist(u)$ then insert (x, u) into $SP(G)$



Algorithm Analysis of InsertEdge(v, w)

Correctness

- If v is affected, then Dijkstra will start at v with the new *dist* values
- All paths of the affected vertices use vertex v
- Since the set of affected vertices is connected, all affected vertices will be inserted into Q and edge scanning will be executed.
- Distances will be calculated correctly (Dijkstra)

Running time

- $T(\text{InsertQ}) = O(1)$
- $T(\text{DecreasePrioQ}() = O(1)$ amortized
- $T(\text{ExtractMinQ}() = O(\log p)$ amortized with p is the number of elements in the heap
- Number of iterations: at most δ
- Total running time: $O(\epsilon + \delta \log \delta)$

Conclusion

Theorem

The algorithm by Ramalingam and Reps provides a bounded dynamic algorithm for the single-source shortest path problem with running time $O(\epsilon + \delta \log \delta)$ *for each update operation* DeleteEdge(v, w) *and* InsertEdge(v, w, c).

Experimental Comparison: Test Setup

Experimental comparison by Demetrescu and Italiano 2006

Random graphs

- Randomly generated graphs with random update sequence

Real-world graphs

- US Road networks: `ftp://edcftp.cs.usgs.gov`
  - 148–952 vertices, 434–2896 edges, weights: $\leq 200\,000$ (distances)
- Internet graphs (AS): `http://www.routeviews.org`
  - 500–3000 vertices, 2406–13 734 védges, weights: $\leq 20\,000$

Bottleneck graphs

- Artificially generated bottleneck graphs: two bipartite components that are connected via one single edge which is updated

Experimental Comparison: Test Setup

- AMD Athlon – 1.8 GHz, 256KB L2 cache, 512MB RAM
- AMD Athlon – 2.1 GHz, 256KB L2 cache, 768MB RAM

- Intel Xeon – 500 MHz, 512KB L2 cache, 512MB RAM
- Intel Pentium 4 – 2.0 GHz, 512KB L2 cache, 2GB RAM

- Sun UltraSPARC IIi – 440 MHz, 2MB L2 cache, 256MB RAM

- IBM Power 4 – 1.1 GHz, 1.4MB L2 cache, 32MB L3 cache, 64GB RAM

Experimental Comparison: Test Setup

- D-RRL: dynamic SSSP by Ramalingam and Reps

- D-KIN: dynamic All-Pairs Shortest Path (APSP) by V. King (1999)
  (This was the first fully dynamic algorithm for SP)

- D-LHP: dynamic APSP-LHP by Demetrescu and Italiano (2004)
  (Locally historical shortest path, $O(n^2 \log^3 n)$ time per update amortized)
- S-LSP: static APSP-LSP by Demetrescu und Italiano (2004)

- S-DIJ: static SSSP by Dijkstra (as base line)

The same data structures for all C-implementations (Heaps, dynamic arrays, Hash maps, graph data structure)

Evaluated Algorithms

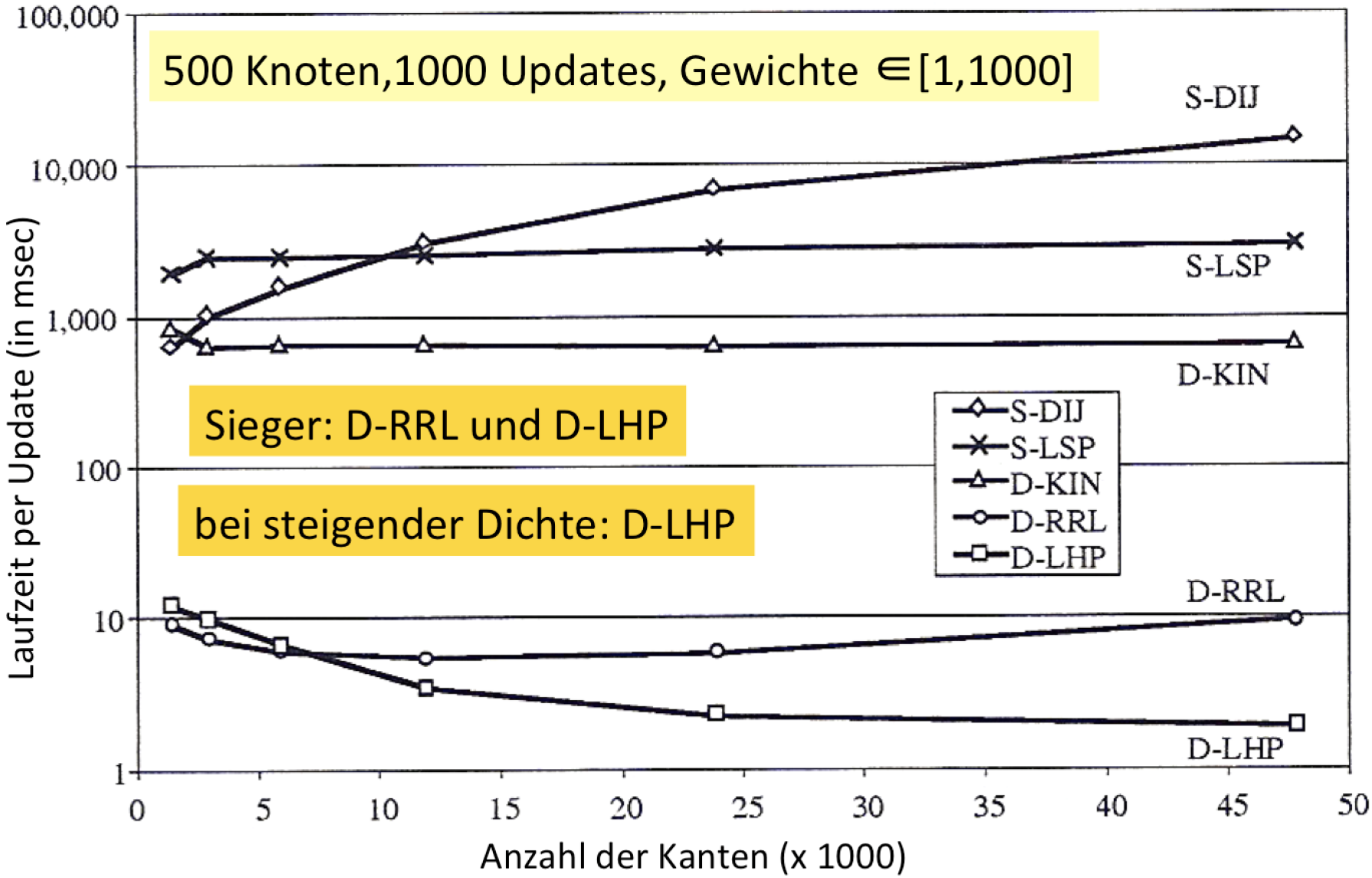
Algorithm	Update time	Query time	Space requirement
S-DIJ	$O(mn + n^2 \log n)$	$O(1)$	$O(n^2)$
S-LSP	$O(\|LSP\| + n^2 \log n)$	$O(1)$	$O(n^2)$
D-RRL	$O(mn + n^2 \log n)$	$O(1)$	$O(n^2)$
D-KIN	$O(n^{2.5}\sqrt{C \log n})$	$O(1)$	$O(n^{2.5}\sqrt{C \log n})$
D-LHP	$O(n^2 \log^3 n)$	$O(1)$	$O(mn \log n)$

C denotes the maximal weight of an edge.

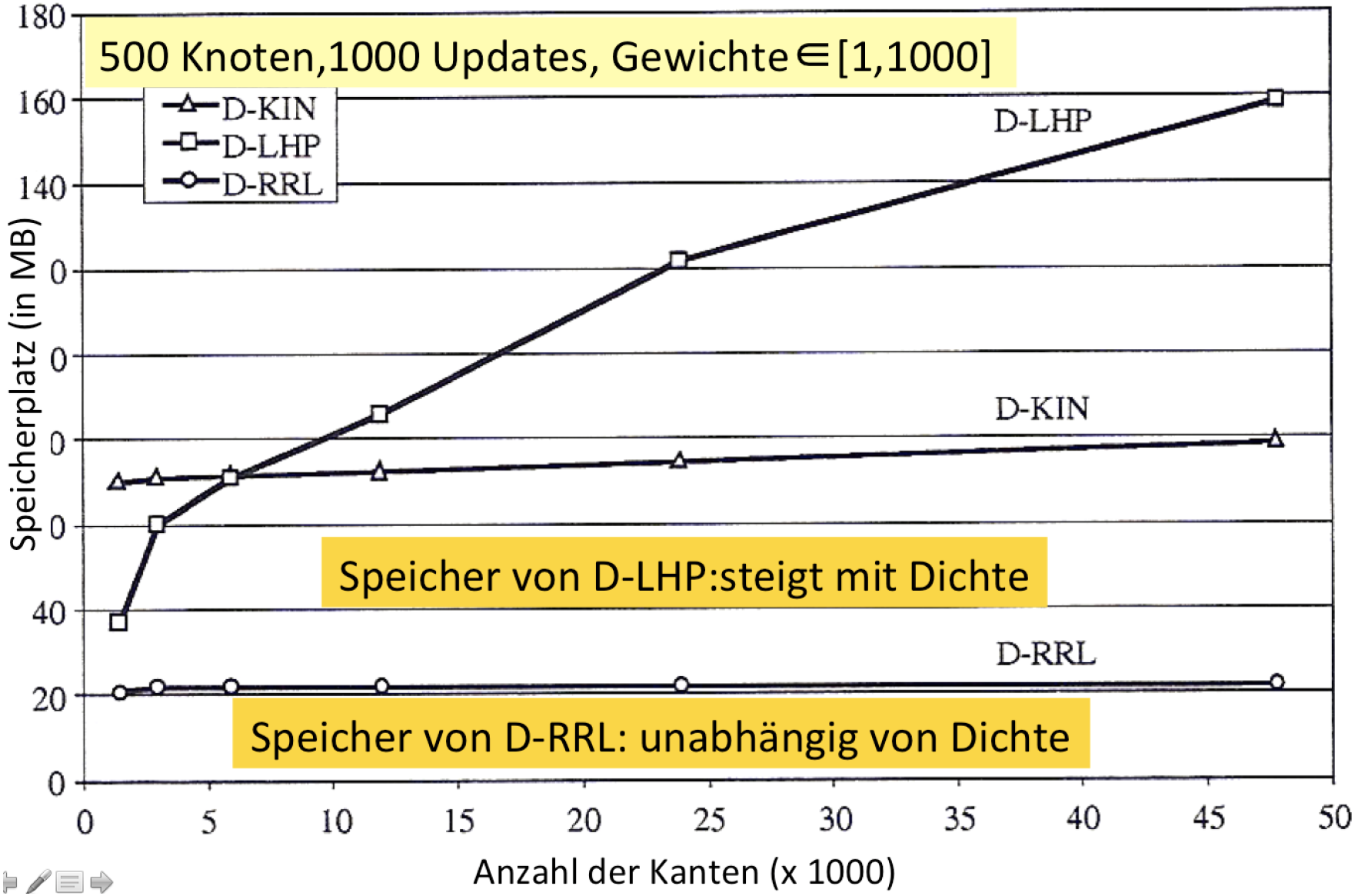
$|LSP|$ is the number of locally shortest paths (relaxing shortest paths by extending it to one more edge) in the graph which could be $O(mn)$ in the worst case.

The authors have also suggested an incremental algorithm (IncreaseOnly) which needs update time $O(n^2 \log n)$ only.

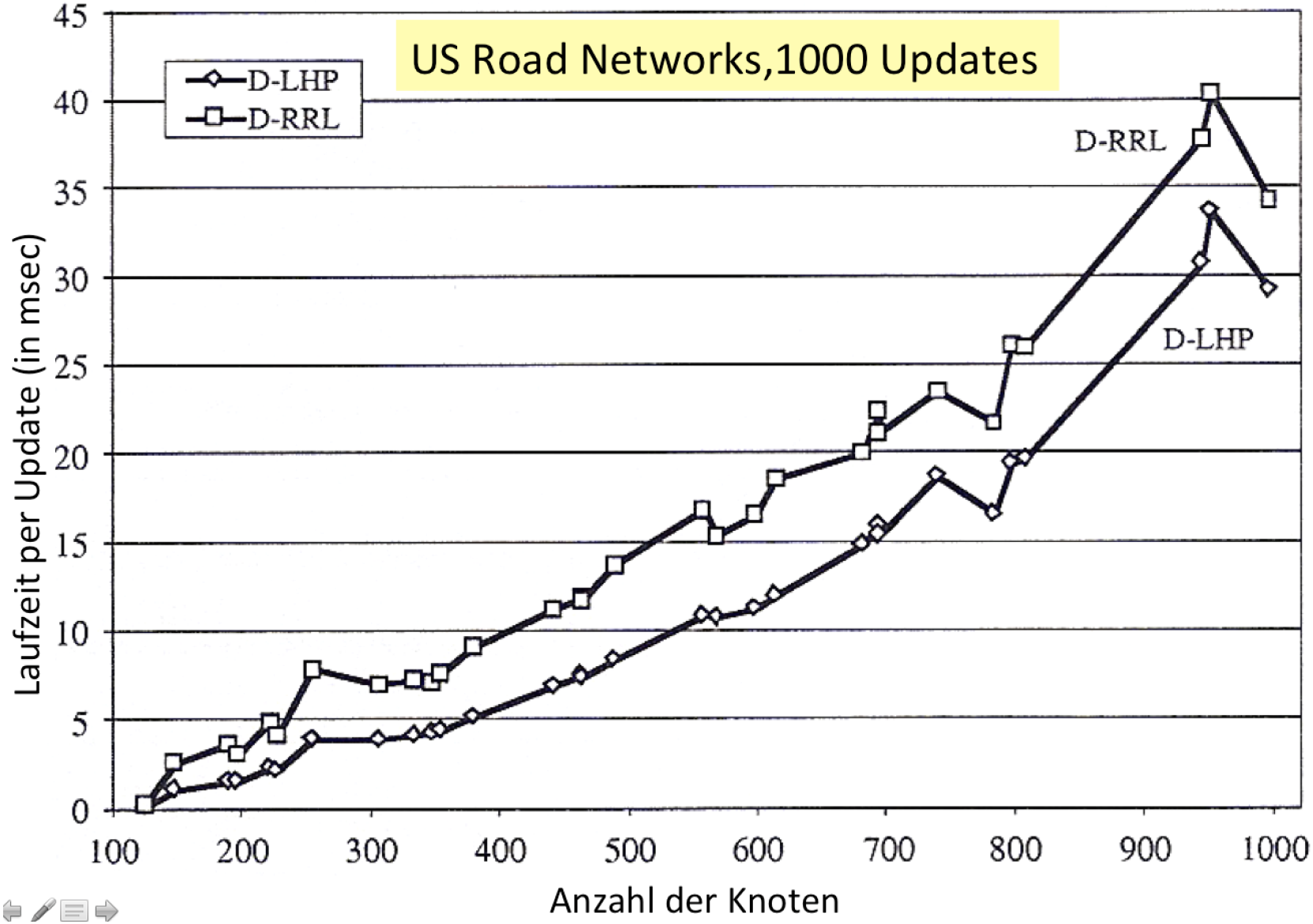
Random Graphs: Running Time per Update



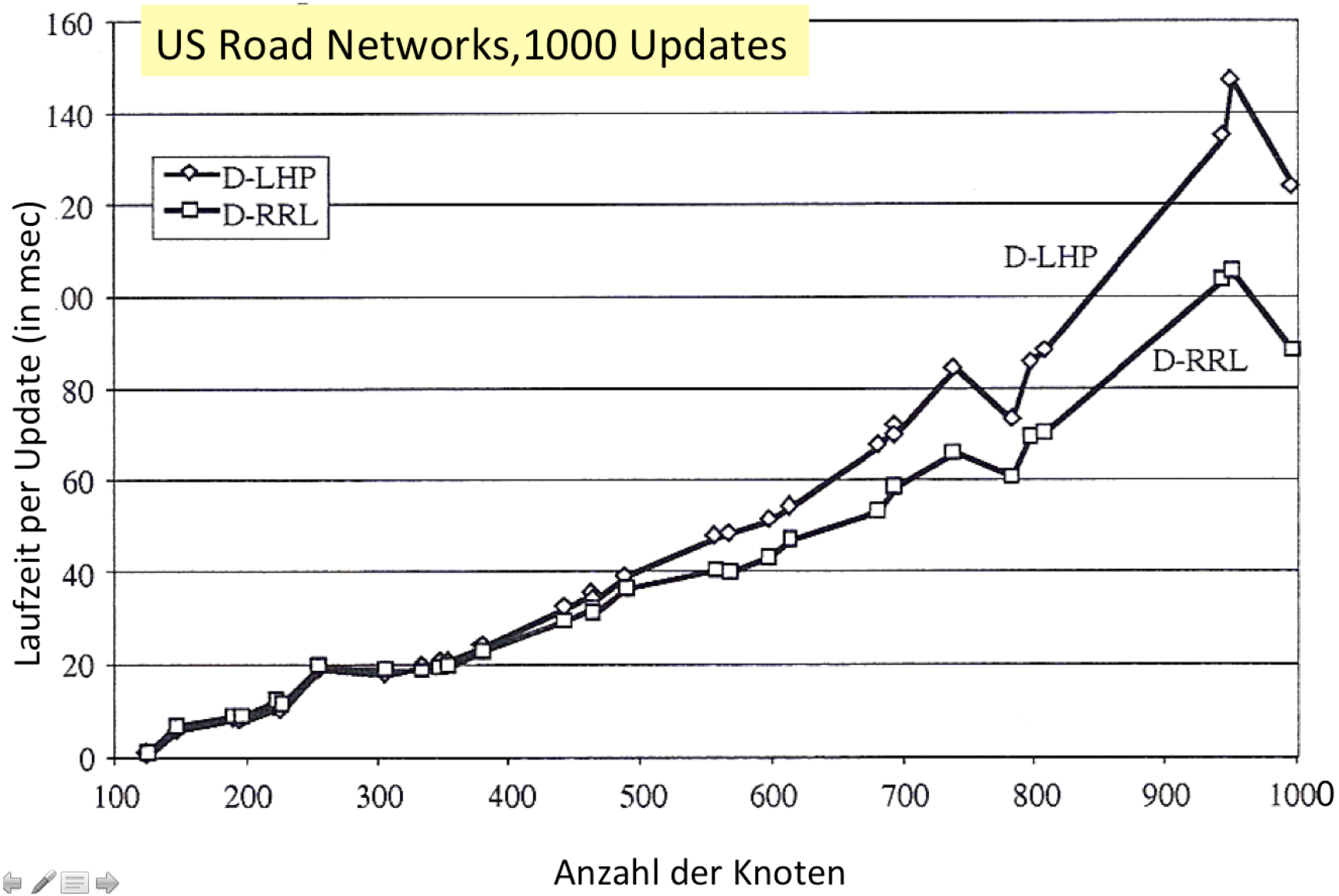
Random Graphs: Space (in MB)



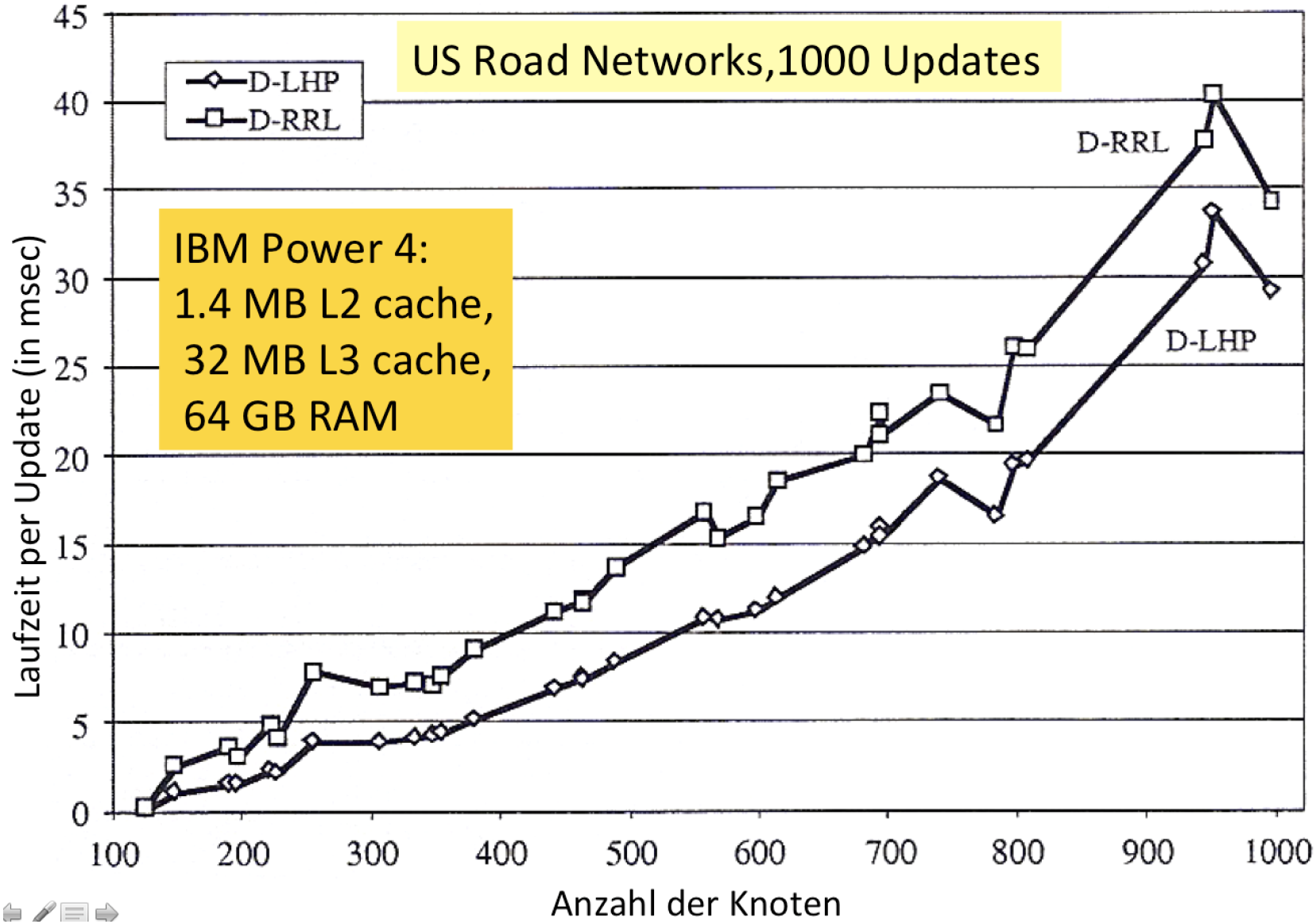
Real World Graphs: Running Time (IBM)



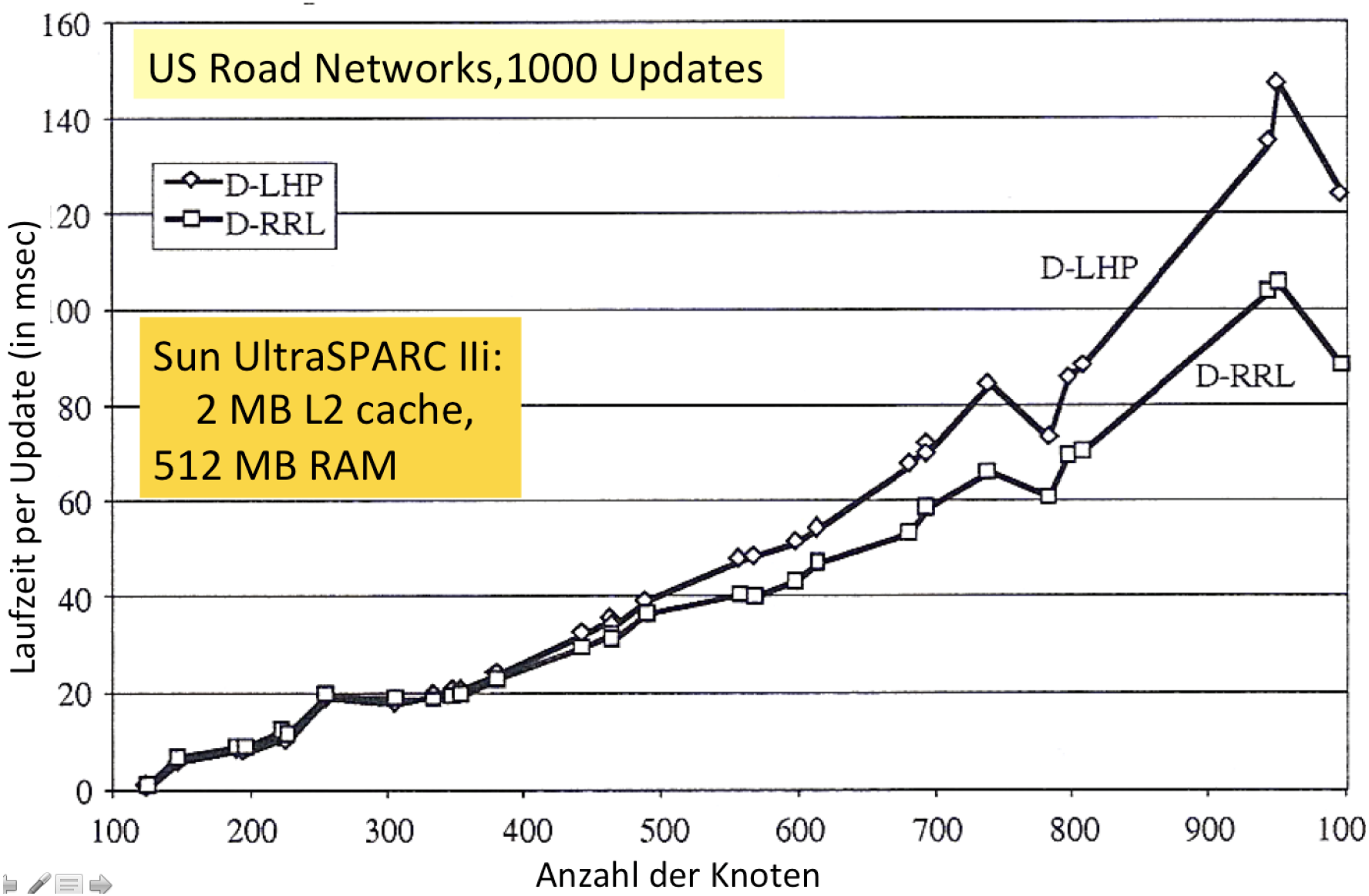
Real World Graphs: Running Time (Sun)



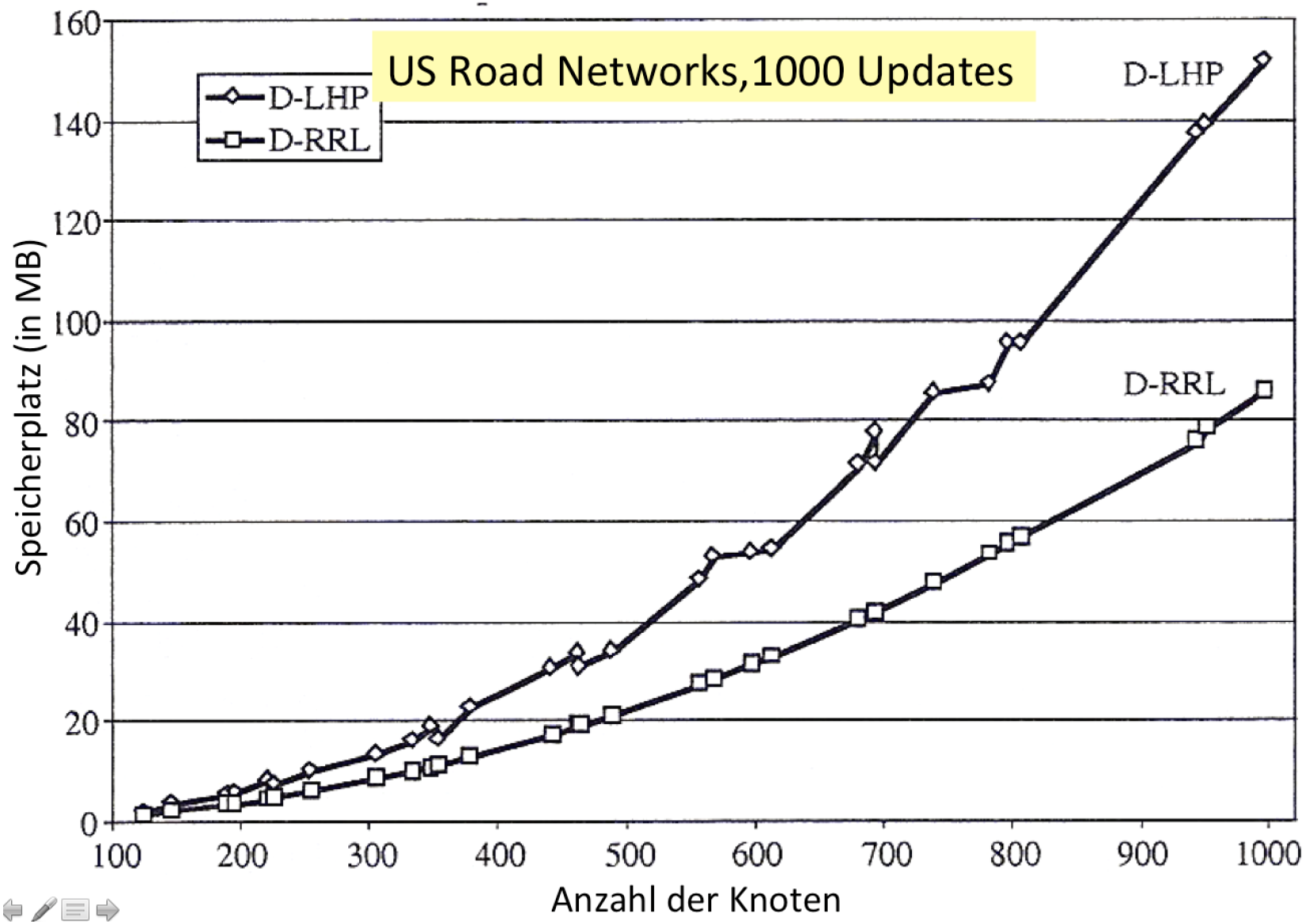
Real World Graphs: Running Time (IBM)



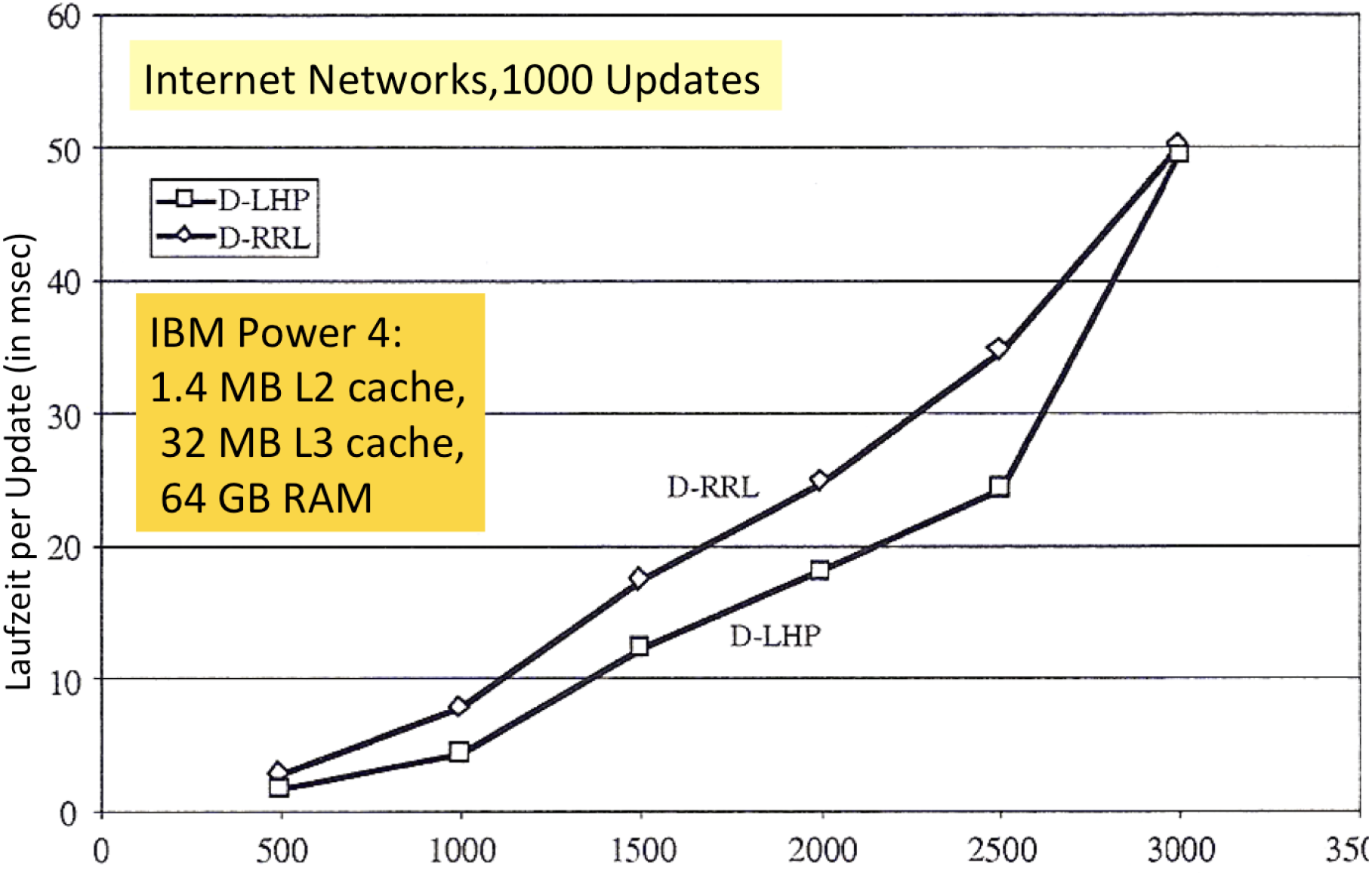
Real World Graphs: Running Time (Sun)



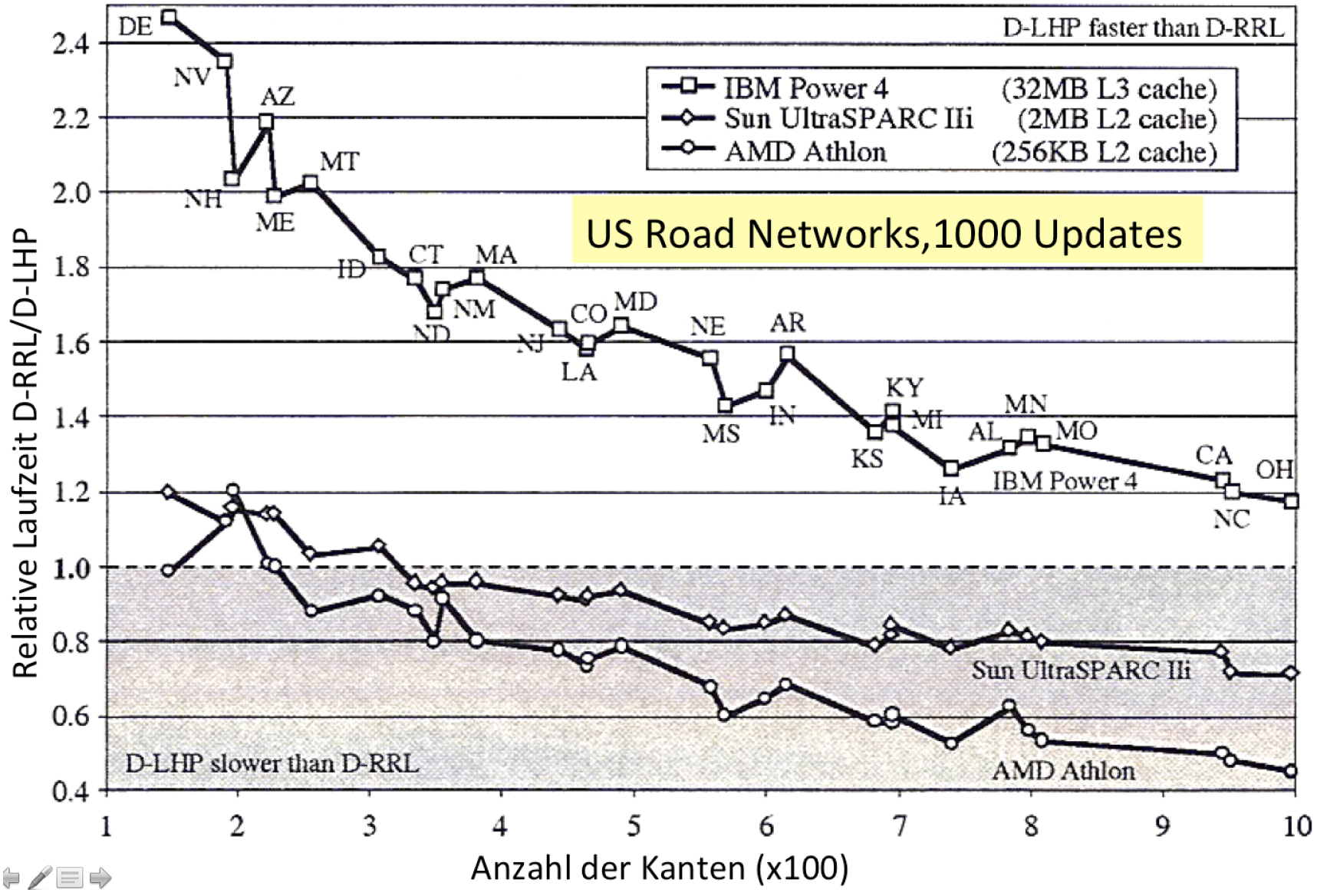
Real World Graphs: Space (in MB)



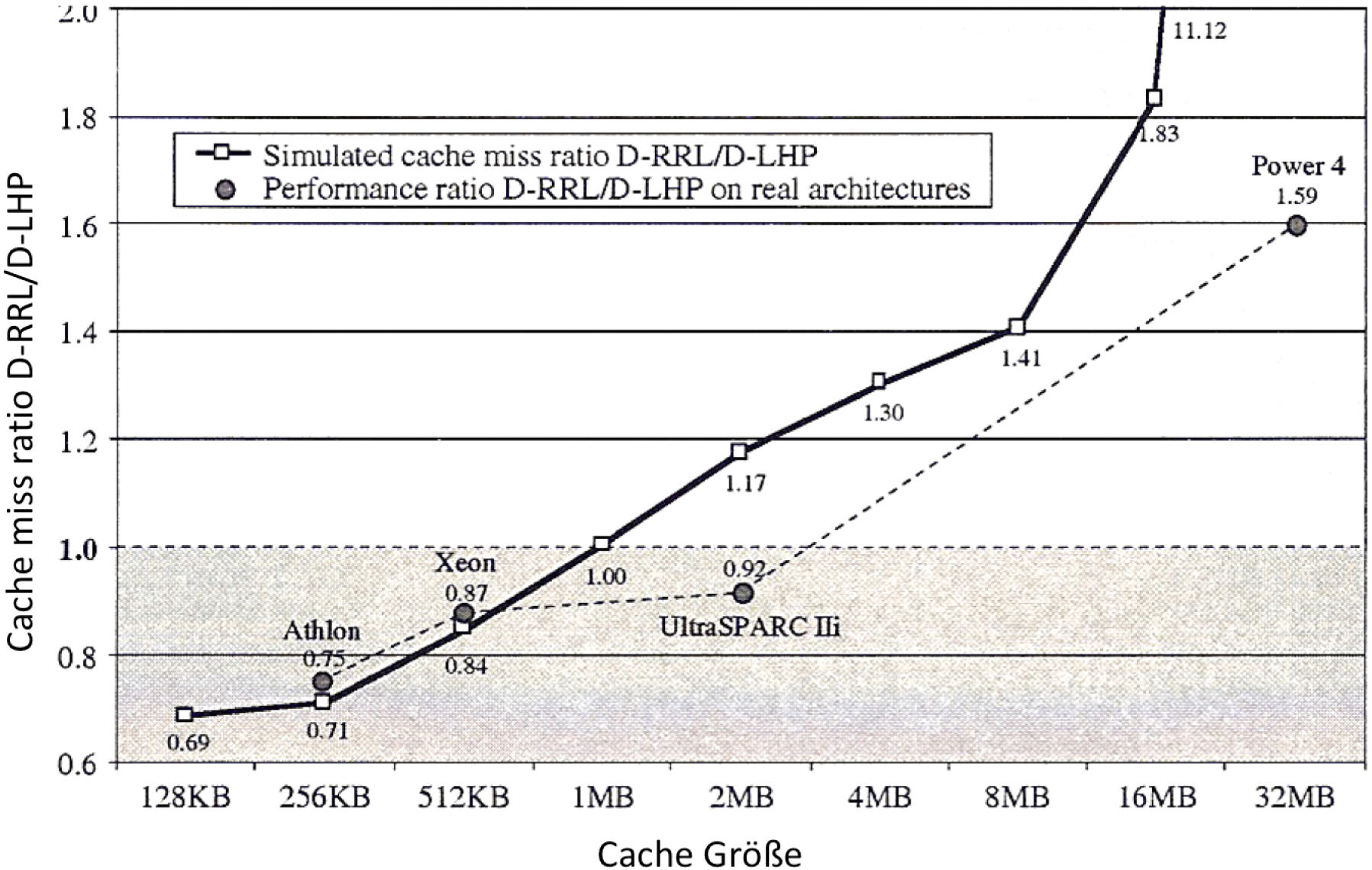
Real World Graphs: Running Time (IBM)



Relative Running Time on various Platforms



Cache Miss Ratio D-RRL / D-LHP



Relative Running Time on Bottleneck Graphs