Algorithm Engineering Reading Notes

6.6050 Algorithm Engineering

September 7, 2025

Contents

A Theoretician's Guide to the Experimental Analysis of Algorithms

Author: David S. Johnson (2001)

1.1 General Notes

1.1.1 Governing Principles

1. **Perform newsworthy experiments**: Problems should have direct applications. Where do you get your test set from if it has no application? Also, algorithms should be somewhat competitive with the ones that are used in practice. Have it be relevant, general, and credible.
2. **Tie paper to the literature**: What actually are the interesting questions? What behaviour needs explaining, and what algorithms seem open to improvement? Try use implementations from previous papers.
3. **Use instance testbeds that can support general conclusions**: You have the choice between (a) instances from real-world/pseudo-real-world applications and (b) randomly generated instances (ideally to be structured like real-world instances). Random instance generators should be able to generate instances of arbitrarily large size.
4. **Use efficient and effective experimental designs**: e.g. variance reduction techniques (use the same set of randomly generated instances for all the algorithms you're testing), bootstrapping to evaluate multiple-run heuristics, use self-documenting programs (save data in descriptively named files)
5. **Use reasonably efficient implementations**: Efficiency comes at a cost in effort. Don't cut corners that would prevent you from making a fair comparison but also don't overdo it.
6. **Ensure reproducibility**: If you run the same code on the same instances of machine/compiler/OS/system load combination you should get the same runtime, operation count, solution quality. More broadly, if someone uses the same method will they be able to draw the same conclusions.
7. **Ensure compatibility**: Write your papers so that future researchers can compare their algorithms/instances to your results
8. **Report the full story**: Don't be overly selective with how you present your data. Maybe include tables in the appendix, but definitely include them.
9. **Draw well-justified conclusions and look for explanations**: What did you learn from your experiments?
10. **Present your data in informative ways**: Use good display techniques!

1.1.2 Possible Questions

1. How do implementation details, parameter settings, heuristics, data structure choices affect runtime of algorithm?
2. How does runtime of algorithm scale with instance size? How does this depend on instance structure?

3. What algorithmic operation counts best to help explain runtime?
4. What in practice are the computational bottlenecks? How do they depend on instance size and structure? How does this differ from predictions of worst-case analysis?
5. How is runtime (and runtime growth rate) affected by machine architecture? Can detailed profiling help explain it?
6. Given that one is running on the same/similar instances and on a fixed machine, how predictable are runtimes?
7. How does runtime compare to top competitors? How are comparisons affected by instance size/structure/machine architecture? Can differences be explained in terms of operation counts?
8. What are the answers to the above questions when "runtime" is replaced by "memory usage"/usage of some other computational resource?
9. What are answers to 1, 2, 6, 7 when one deals with approximation algorithms and "runtime" is replaced with "solution quality"?
10. Given a new class of instances you've identified, does it cause significant changes in the behaviour of previously studied algorithms?

1.1.3 Whom can you trust?

- Never trust a random number generator
- Never trust your code to be correct
- Never trust a previous author to have known all the literature
- Never trust your memory as to where you put that data (and how it was generated)
- Never trust your computer to remain unchanged
- Never trust backup media or websites to remain readable indefinitely
- Never trust a so-called expert on experimental analysis

1.2 Paper Review

1.2.1 Motivation

Describes issues that arise when algorithms are analyzed experimentally (challenges with rigorous analysis led to the emphasis on theoretical worst-/average-case analysis w.r.t. asymptotic behavior in the first place)

1.2.2 Key Ideas/Results

- Key metrics are resource usage (time/memory) and quality of output solution
- Reasons for implementing an algorithm:
  - **Application paper**: Using code in a particular application, often to prove mathematical conjectures. Here, result is more important than efficiency
  - **Horse-race paper**: Evidence of superiority of algorithmic ideas
  - **Experimental analysis paper**: Understand strengths, weaknesses, operation of interesting algorithmic ideas in practice
  - **Experimental average-case paper**: Understand average-case behaviour of algorithms where direct probabilistic analysis is too hard

- Rules for governing the writing of experimental papers:
  - Perform newsworthy experiments
  - Tie paper to literature
  - Use instance testbeds that can support general conclusions
  - Use efficient and effective experimental designs
  - Use reasonably efficient implementations
  - Ensure reproducibility
  - Ensure comparability
  - Report the full story
  - Draw well-justified conclusions and look for explanations
  - Present your data in informative ways

1.2.3 Pet Peeves

- Authors/referees who don't do their homework (make sure your algorithm isn't dominated)
- Focusing on unstructured random instances that don't reflect real data OR on exclusively real data that may be outdated
- In the millisecond testbed, runtime is somewhat irrelevant
- When evaluating approximation algorithms, don't limit yourself to algorithms where the solution/optimal value is known (in this case, using an optimization algorithm is reasonable, so why use approximation)
- Claiming "inadequate programming time/ability" as an excuse
- Supplying code that doesn't match a paper's description of it
- Irreproducible standards of comparison (e.g. only reporting the solution value, only reporting the percentage excess over the best solution currently known, reporting percentage excess over an estimate of expected optimal for randomly generated instances, reporting percentage excess over a well-defined lower bound, reporting percentage excess/improvement over some other heuristic)
- Using runtime as a stopping criterion (it's not a cake that you're baking)
- Using optimal solution value as a stopping criterion for algorithms that have no way of verifying optimality
- Hand-tuned algorithm parameters
- One-run study (unless study covers a wide range of instances)
- Using the best result found as an evaluation criterion (is often from the tail of the distribution)
- Uncalibrated machine (include processor speed, operating system, language/compiler)
- The lost testbed (make sure future researchers have access to your tests)
- False precision (more digits of accuracy than justified by the data)
- Unremarked anomalies
- Ex post facto stopping criterion (total running time is not reported, just the time taken to find the answer)
- Failure to report overall runtimes (even if your main focus is not runtime)

- Data without interpretation (at least be able to summarize patterns in the data)
- Conclusions without support
- Myopic approaches to asymptopia (medium/large instance behaviour doesn't necessarily translate to VERY large instance behaviour)
- Tables without pictures
- Pictures without tables
- Pictures with too little insight
- Inadequately/confusingly labeled pictures
- Pictures with too much information (best when clear and uncluttered)
- Confusing pictorial metaphors
- Spurious trend lines (drawing lines between each point)
- Poorly structured tables (order rows and columns so that they highlight important information)
- Undefined metric (cryptic labels)
- Comparing apples to oranges (algorithms tested on different instances, or on different machines)
- Detailed stats on unimportant questions
- Comparing approximation algorithms as to how often they find optima (restricts attention to test instances for which optimal solutions are known, ignores question of how near to optimal algorithm gets when it does not find the optimum)
- Too much data! Replace raw data with averages and other summary stats

1.2.4 Pitfalls

- Dealing with dominated (always slower than status quo) algorithms
- Devoting too much computation to the wrong questions (overstudying results, running full experimental suites before algorithm is efficient or you've decided what data to collect)
- Getting into an endless loop in the experimentation (draw the line on future research)
- Using randomly generated instances to evaluate behaviour of algorithm ends up exploring properties of randomly generated instances
- Too much code tuning (don't overthink, concentrate programming effort where it will be the most useful)
- Lost code/data (don't modify code without saving version of original, don't forget to keep backup copies, organize your directories)

1.2.5 Suggestions

- Think before you compute. Have you implemented it correctly? What do you want to study? What are your experiments addressing?
- Use exploratory experimentation to find good questions
- Use benchmark codes to calibrate machine speeds (some portable source code that other researchers can use to calibrate in the future)
- Use profiling to understand runtimes (number of calls to various subroutines, time spent in subroutines etc. can point out higher-exponent components of runtime earlier). Also offers a mode of explaining the results.
- Display normalized runtimes

1.2.6 Novelty

Very experienced researcher discussing the state of the art of paper writing in this (somewhat niche?) field.

1.2.7 Strengths/Weaknesses

Personal/objective opinion. Otherwise very thorough and goes through a whole series of do's/don'ts that are akin to the advice a PI might give when instructing students on how to write a paper in that field.

1.2.8 Ideas for Improving Techniques/Evaluation

1.2.9 Open Problems/Directions for Future Work

Would be nice to have some sense of the appropriate structure.

Algorithm Engineering: An Attempt at Definition

Author: Peter Sanders

2.1 General Notes

2.1.1 Introduction

- Fast search (Google) makes large quantities of text searchable
- Algorithms offer efficiency + performance guarantees
- Gap between theory and practice:
  - Traditional machine models for theorizing about algorithms don't match hardware parallelism, memory hierarchies etc.
  - Theoretically designed algorithms often not implementable
- Algorithmic engineering process similar to scientific method:
  - Falsifiable hypothesis that can be (in)validated by experiments
  - Reproducibility
  - Application-oriented design supplies realistic inputs, constraints, and design parameters
- Distinct from application engineering (which sits at the intersection of algorithm engineering and software engineering/developing production quality code):
  - Algorithm engineering emphasises fast dev, efficiency, instrumentation for experiments
  - Application engineering emphasises simplicity, thorough testing, maintainability, simplicity, tuning for particular inputs
  - Link is often created through algorithm libraries

2.1.2 History

- 1970s/80s: Algorithm theory become subdiscipline of CS devoted to "paper and pencil" work
- 1986: "Algorithm engineering" is term but not discussed
- 1997: "Workshop in Algorithm Engineering" developed

2.1.3 Models

2.1.4 External Memory Model (I/O Model)

- Two levels of memory (c.f. uniform memory): fast memory of limited size M and slow memory accessed in blocks of size B
- Cost to doing internal work (cost ratio between disk memory and RAM is $\sim$ 1:200)

2.1.5 Design

- Efficient algorithms must also consider constant factors
  - e.g. Maximum Flow algorithms: asymptotically best algorithm performs much worse than theoretically inferior algorithms

2.1.6 MST Example

- Undirected connected graph G with n nodes and m edges (edges with nonnegative weights)
- MST of G is subset of edges with minimum total weight that forms a spanning tree of G
- Can be solved in O(sort(m)) expected I/O steps where sort(N) = O(N/B $\log_{M/B}$ N/B) and denotes the number of I/O steps required for external sorting.
- New algorithm – semiexternal variant of Kruskal's algorithm:
  - Semiexternal graph is allowed O(n) words of fast memory
  - Edge accepted into MST if connecting two components of forest defined by previously found MST edges (does not create a cycle). Uses union-find (disjoint set union, DSU) data structure to store connected components
  - m ¿ what can fit into memory but n can. Edges are sorted externally and then streams them into RAM to apply Kruskal's procedure there. Using path compression, only log(n) bits are needed per node.

2.1.7 Analysis

- Difficult to analyze even some simple/proven algorithms
- Simplex algorithm is exponential (theoretically) but in practice, and especially with the addition of random permutations, linear/polynomial

2.1.8 Implementation

- Essentially describes several algorithms that were too complex/required too difficult data structures to implement efficiently until many years later

2.1.9 Experiments

- Benchmarking the falsifiable hypothesis
- Difficult to test experiments with external memory algorithms (large inputs, huge runtime)

2.1.10 Algorithm Libraries

- Challenging to design since applications are "unknown" at implementation time
- Interface needs to be distinct from implementation
- Examples:
  - LEDA (Library of Efficient Data Types and Algorithms) for C++
  - Boost: forum for library designers that ensures quality and maybe to become part of STL
  - CGAL (Computational Geometry Algorithms Library): sophisticated example of C++ template programming

2.1.11 Instances

- Collections of realistic problem instances for benchmarking NP-hard problems e.g. traveling salesman, Steiner tree, satisfiability, graph partitioning
- More difficult to obtain polynomial ones e.g. route planning, flows etc. (which usually use random inputs instead)

2.2 Paper Review

2.2.1 Motivation

"Algorithmic engineering" lacks a formal definition but has a clear use-case. Paper is attempt at defining the field.

2.2.2 Key Ideas

- Algorithm engineering is the experimental branch of algorithmic theory
  - P. Italiano "Workshop in Algorithm Engineering" (1997) describes algorithm engineering as "concerned with the design, analysis, implementation, tuning, debugging, and experimental evaluation of computer programs for solving algorithmic problems... provides methodologies and tools for developing and engineering efficient algorithmic codes and aims at integrating and reinforcing traditional theoretical approaches for the design and analysis of algorithms and data structures"
- We need new memory models to capture the internal work done by an algorithm

2.2.3 Results

Uses Minimum Spanning Tree (MST) algorithm as an example.

2.2.4 Novelty

2.2.5 Strengths/Weaknesses

2.2.6 Ideas for Improving Techniques/Evaluation

2.2.7 Open Problems/Directions for Future Work

Parallel Algorithms

Authors: Guy E. Blelloch and Bruce M. Maggs

3.1 General Notes

- Parallelism in an algorithm $\neq$ ability of a computer to perform operations in parallel

3.2 Modeling Parallel Computations

- Sequential algorithms formulated using RAM (random access machine) model i.e. one processor connected to a memory system
  - Each basic CPU operation (arithmetic, logic, memory access) takes one time step
- Parallel computers have more complex organization

3.2.1 Multiprocessor Models

- Sequential RAM with more than one processor
- Types: local memory machine models, modular memory machine models, parallel random-access machine (PRAM) models
  - **LMM**: One interconnection network with many processors. Each processor has its own local memory. Local operations (e.g. local memory access) take unit time. Time taken to access another processor's memory is larger.
  - **MMM**: Interconnection network has many processors and many memories. Processors are linked to memory via interconnection network, and access it using memory requests through the network. Time for any processor to reach any given memory module is approximately uniform.
  - **PRAM**: One shared memory with many processors. A processor can access any "word" of memory in one step. Accesses from multiple processors can happen in parallel. "Ideal" case, not realistic.

3.2.2 Network topology

Definition 1. Network: collection of switches connected by communication channels. Processor/memory module has 1+ communication ports connected to the switches.

Definition 2. Network topology: pattern of interconnection of switches.

Definition 3. 2-dimensional mesh: network that can be laid out in a rectangular fashion. Each switch has label (x, y) i.e. its coordinates within X, Y bounds.

Algorithms have been designed to work with specific types of memory. May not work well on other networks and more complex than those designed for abstract models e.g. PRAM.

- Bus: simplest network topology
  - Local and modular memory machine models
  - All processors/memory modules are connected to a single bus

  - At most one piece of data can be written to the bus per step (request from processor to read/write or response from processor/memory module that holds the value)
  - Pros: easy to build, all processors/memory modules can observe traffic on the bus so easy to develop protocols for local cacheing
  - Cons: processors have to take turns using the bus (especially a problem with many processors)
- Mesh:
  - Number of switches = $X \cdot Y$
  - Often in local memory machine models (each processor with its local memory is connected to each switch)
  - Remote memory accesses done by routing messages through the mesh
  - Can also have 3D meshes, torus meshes, hypercubes
- Multistage network:
  - Used to connect one set of input switches to another set of output switches through a sequence of stages of switches
  - Stages numbered 1-L (L = depth of network). Input switches are on stage 1, output on L.
  - Often used in modular memory computers (processors attached to input, memory to output). Word of memory accessed by injecting memory access request message into network.
  - 2-stage network connecting 4 processors to 16 memory modules. Each swtich has two channels at bottom and four at the top. More memory than processors because processors can generate memory access requests faster than memory modules can service them
- Fat-tree:
  - Structured like a tree. Each edge can represent many communication channels/each node can represent many switches.
  - Memory requests travel down the tree to least-common ancestor then go back up.

Alternate modeling technique: latency and bandwidth

- Latency: time taken for message to traverse the network (topology dependent)
- Bandwidth: rate at which data can be injected into the network (topology dependent); minimum gap g between successive injections of messages into the network.
- Models include Postal Model (model described by single parameter L), Bulk-Synchronous Parallel model (L and g), LogP model (L, g, and o (overhead/wasted time on sending/receiving message))

3.2.3 Primitive operations

What kinds of operations are the processors/network able to perform?

- Assume all porcessors are allowed to perform in same local instructions as single processor in RAM
- Special instructions for non-local memory requests, sending messages to other processors, global operations e.g. synchronization, restrictions to avoid processors from interfering with each other (e.g. writing to same memory locaiton)
- Nonlocal instruction types:
  - Instructions performing concurrent accesses to same shared memory locaiton
  - Instructions for synchronization
  - Instructions performing global operations on data

- What happens when processors try to read/write to same resource (processor, memroy module, memory location) at the same time?
  - Exclusive access to resource: forbid the action
  - Concurrent access to resource: unlimited access
  - Queued access: time for a step is proportional to the max number of accesses to a resource
- Other primitives support synchronization, combining arithmetic operations with communication etc.

3.2.4 Work-Depth Models

Definition 4. Work depth model: the cost of an algorithm is determined by examining the total number of operations it performs (W), and the dependencies among those operations (D).

Definition 5. W work: Number of operations an algorithm performs D depth: longest chain of dependencies among operations P paralellism: W/D

If the sequential cost of an algorithm is W, then the ideal parallel time with P processors is $\approx T_P \approx W/P + D$ where we can evenly spread work amongst processors but cannot go faster than the critical path.

- Pros: no machine-dependent details. Often lead to realistic implementation
- Classes:
  - Circuit models: most abstract. Nodes and directed arcs, where node is a basic operation. Number of incoming arcs = fan-in, outgoing arcs = fan-out. Input arcs provide input to the whole circuit. No directed cycle permitted.
  - Vector machine models: algorithm is a sequence of steps which perform operations on a vector of input values, producing an output vector.
  - Language-based models: work-depth cost is associated with each programming language construct (e.g. work of calling two functions in parallel == work of two calls)

3.2.5 Costs

Work W = number of processes $\times$ time required for algorithm to complete execution Depth D = total time required to execute the algorithm

3.2.6 Emulations

An algorithm designed for one parallel model can often be translated into algorithms that work for another (work-preserving, i.e. work performed by both algorithms is approximately the same)

This section goes through a proof showing that a PRAM processor can be "emulated" by a more realistic multiprocessor processor with a butterfly network with only logarithmic slowdowns.

3.3 Parallel algorithmic techniques

3.3.1 Divide-and-conquer

Split the problem into subproblems that are easier to solve than the original, solve the subproblem, and merge the solutions. Often inherently parallelizable as operations are typically independent.

Mergesort

- Sorts n keys by splitting keys into two sequences of n/2 keys, recursively sorting each sequence, merging two sorted sequences of n/2 keys into sorted sequence of n keys.
- Each half can be sent to a parallel process

3.3.2 Randomization

- Used in parallel algorithms to ensure that processors can make global decisions which very probably lead to good global decisions. E.g. selecting a representative sampling, breaking symmetry, load balancing (dividing data into evenly sized subsets)

3.3.3 Parallel pointer manipulation

Many operations for lists, trees, graphs (e.g. traversing linked list, visiting tree nodes, DFS) are inherently sequential. Parallel alternatives include:

- **Pointer jumping** for lists and trees. Each node in parallel replaces its pointer with that of its successor/parent. After (at most log n) steps every node points to the root of the tree
- **Euler tour** computing subtrees, tree depths, or node levels that would originally require sequential traversal can be done all at the same time
- **Graph contraction** a graph is reduced in size while maintaining some of its original structure. Problem is solved on contracted graph, then used for final solution.
- **Ear decomposition** Partition of graph edges into ordered collection of paths. First is a cycle, others are ears. Replaces depth first search.
- Other: small grpah seperators, hashing for load balancing and mapping addresses to memory

3.4 Basic operations on sequences, lists, trees

```
1 ALGORITHM: sum(A)
2 if (A.size() = 1) return A[0]
3 else return sum({A[2i] + A[2i + 1] : i \in [0...A.size()/2]})
```

This can also be used to calculate max etc.

```
1 ALGORITHM: scan(A)
2 if (A.size() == 1) return [0]
3 else
4     S = scan({A[2i] + A[2i + 1] : i in [0..A.size()/2]})
5     R = {if (i mod 2 == 0) then S[i/2] else S[(i-1)/2] + A[i-1] : i in [0...A.size
          ()]}
```

Element-wise adding even elements of A to the odd elements, recursively solving the problem on the resulting sequence.

Multiprefix generalizes the scan operation to multiple independent scans, where for [(1,5), (0,2), (0,3), (1,4), (0,1), (2,2)] each position receives the sum of all the elements that have the same key to yield [0, 0, 2, 5, 5, 0].

Fetch and add is multiprefix but the order of input elements for the scan is not necessarily the same as the order in input sequence A.

3.4.1 Pointer jumping

Each node i replaces pointer P[i] with pointer of the node it points to, P[P[i]]. Can compute a pointer to the end of the list/root of tree for each node.

```
1 ALGORITHM point_to_root(P)
2 for j from 1 to [log(P.size())]
3     P := {P[P[i]] : i \in [0...P.size()]}
```

3.4.2 List ranking

Computing distance from each node to the end of a linked list.

```
ALGORITHM list_rank(P)
V = {if P[i] = i then 0 else 1 : i \in [0...P.size()]}
for j from 1 to [log(P.size())]
    V := {V[i] + V[P[i]] : i \in [0...P.size()]}
    P := {P[P[i]] : i \in [0...P.size()]}
```

Where V[i] is distance spanned by pointer P[i] w.r.t. original list.

These are not work efficient since it takes O(n log n) work vs sequential algorithms that can do it in O(n).

3.4.3 Removing duplicates

Input and output are both sequences. Order doesn't matter.

1. Using an array of flags: initialize a second array that keeps track of the initial appearance of each value. Only add the ones that do not repeat more than once. Explodes for longer lists.

2. Hashing: create a hash table containing a prime number of entries, where prime is 2x as big as the number of items. If multiple items are attempted to be written into the hash table, only one will succeed. May not work the first iteration because values are defeated by values that are different. Needs several iterations with different hash functions.

```
ALGORITHM remove_duplicates(V)
m := next_prime(2 * V.size())
table := distribute(-1, m)
i := 0 // different hash function used for each iteration
result := {}
while V.size() > 0
    table := table <- {(hash(V[j], m, i), j) : j \in [0...V.size()]}
    winners := {V[j] : j \in [0...V.size()] | table[hash(V[j], m, i)] = j}
    result := result ++ winners
    table := table <- {(hash(k, m, i), k) : k \in winners}
    V := {k \in V | table[hash(k, m, i)] != k}
    i := i + 1
return result
```

3.5 Graphs

Most graph problems do not parallelize well.

Definition 6. Sparse graph: m (number of edges) $\ll n^2$ (number of nodes)

Definition 7. Diameter D(G): maximum, over all pairs of vertices (u,v), of the minimum number of edges that need to be traversed from u to v

Edge lists, adjacency lists, adjacency matrices used to represent graphs. For parallel algorithms, linked lists are reprsented with arrays (e.g. edge-list = array of edges, adjacency-list = array of arrays).

3.5.1 BFS

Parallel similar to sequential version. Start with source vertex s, traverse each level of the graph to find vertices that have not yet been visited. Each level is visited in parallel and no queue is required.

Maintain a set of frontier vertices to keep track of current level and produce new frontier on next step. Collect all neighbours of current frontier vertices in parallel and remove any that have not been visited. Multiple vertices may have the same uncollected vertex, so we need to remove duplicates.

```
ALGORITHM: bfs(s, G)
front := [s]
tree := distribute(-1, G.size())
tree[s] := s
while (front.size() != 0)
    E := flatten({{(u,v) : u \in G[v]} : v \in front})
    E' := {(u,v) : E | tree[u] = -1}
    tree := tree <- E'
    front := {u : (u,v) \in E' | v = tree[u]}
return tree
```

Where front is the frontier vertices, tree contains the current BFS tree. Iterations terminate when no more vertices in frontier. Each vertex and edges are only visited once, so O(m+n).

Can generate trees that cannot be generated with sequential BFS.

3.5.2 Connected components

How to label connected components of an undirected graph, such that two vertices u, v have the same label if and only if there is a path between them. BFS and DFS are very inefficient.

Graph contraction helps. Contract vertices of a subgraph into a single vertex.

1. Random mate graph contraction

- Form a set of star subgraphs (tree depth 1), contract the seperators. Merge child into parent
- Randomly decide if vertex is parent or child. Neighboring parent vertex is identified and made the child's root.
- Repeat until all components have size 1

How many contraction steps we need depends on how many vertices are removed in each step. Only children will be removed (P(child) = 1/2), and only if there's a neighbouring parent (P(has neighbouring parent) ¿= 1/2). The probability that a vertex is removed is P = 1/2 * 1/2 = 1/4.

```
ALGORITHM cc_random_mate(labels, E)
if E.size() == 0 return labels
else
    child := {rand_bit() : v \in [1...n]} // randomly become child/parent
    hooks := {(u,v) \in E | labels[u] != labels[v]} // edges with different labels
    labels := labels <- hooks // children adopt parent label through hooks
    E' := {(labels[u],labels[v]) : (u,v) \in E | labels[u] != labels[v]} // create
        smaller graph
    labels' := cc_random_mate(labels, E') // recursively solve on smaller graph
    labels' := labels <- {(u,labels'[v]) : (u,v) \in hooks} // propagate labels
        back through hook
return labels'
```

The re-expansion of the graph passes labels from each root of a contracted star to its children. The graph is contracted while going down recursion and re-expanded coming back up. Each child can have multiple hook edges but only one parent. For each hook edge, the parent's label is written into the star.

Coins are flipped on even already-contracted vertices.

Possible improvements: don't use all the edges for hooking on each step, just use a sample.

2. Deterministic graph contraction Form a set of disjoint subgraphs (each is a tree). Use point-to-root algorithm to contract subgraphs to a single vertex.

```
ALGOIRTHM cc_tree_contract(labels, E)
if (E.size() == 0) return labels
else
    hooks := {(u,v) \in E | u > v}
    labels := labels <- hooks
```

```
    labels := point_to_root(labels)
    E' := {(labels[u], labels[v]) : (u,v) \in E | labels[u] != labels[v]}
    return cc_tree_contract(labels, E')
```

Instead of parent/child, the hooks are selected by using pointer jumping. Point goes from larger numbered vertices to smaller numbered vertices. Worst case behaviour occurs when the maximum index is at the center of a star and all of its children are smaller than it is.

Possible improvements: interleave hooking steps with pointer-jumping steps. Tree is only partially contracted when executing each hooking step.

3.5.3 Spanning trees and minimum spanning trees

Definition 8. Spanning tree of connected graph G = (V,E) is connected graph T = (V,E') s.t. E' is a subset of E, and E'.size() = V.size() - 1. Cannot have any cycles and forms a tree.

When components are hooked together algorithm can keep track of which edges were used. Collection of all edges used in hooking (since only used once) will form a spanning tree.

For minimum spanning tree, random mate algorithm makes sure that it selects the minimum-weight edge. If it doesn't lead to a parent, the child does not connect and is orphaned.

3.6 Sorting

Focusing on two algorithsm: QuickSort and radix sort.

3.6.1 QuickSort

```
ALGORITHM: quicksort(A)
if A.size() == 1 then return A
i := rand_int(A.size())
p := A[i]
in parallel do
    L := quicksort({a : a \in A | a < p}) \\ less than
    E := {a : a \in A | a == p} \\ equal to
    G := quicksort({a : a \in A | a > p}) \\ greater than
return L ++ E ++ G
```

Can be further parallelized by selecting more than one partition element. With P processors, choosing P-1 partition elements divides keys into P sets, each of which can be sorted. Make sure to assign same number of keys per processor.

3.6.2 Radix sort

Not a comparison sort (does not compare keys directly to find relative ordering); instead, reprsents keys as b-bit integers.

Examines keys to be sorted one digit position at a time, starting with the least significant digit in each key. The output ordering of this sorrt must preserve the input order of any two keys with identical digit values in the position being examined.

Counting sort finds the rank of each key (position in the output order) then permutes the keys.

```
ALGORITHM radix_sort(A,b)
for i from 0 to b-1
    flags := {(a >> i) mod 2 : a \in A}
    notflags := {1-b : b \in flags}
    R0 := scan(notflags)
    s0 := sum(notflags)
    R1 := scan(flags)
    R := {if flags[j] == 0 then R0[j] else R1[j] + s0 : j \in [0...A.size()]}
```

```
    A := A <- {(R[j],A[j]) : j \in [0...A.size()]}
return A
```

Can also handle floating point numbers.

3.7 Computational geometry

Calculating properties of sets of objects in k-dimensional space (e.g. closest-pair of points, convex-hull, line/polygon intersections).

Parallel solutions often use divide-and-conquer or plane sweeping.

3.7.1 Closest pair

Set of points in k dimensions, returns two points that are closest to each other (Euclidean). Uses divide and conquer to split points along lines parallel to the y-axis.

```
ALGORITHM closest_pair(P)
if (P.size() < 2) return (P, inf)
x_m := median({x : (x,y) \in P})
L := {(x,y) \in P | x < x_m}
R := {(x,y) \in P | x > x_m}
in parallel do
    (L',delta_l) := closest_pair(L)
    (R',delta_r) := closest_pair(R)
P' := merge_by_y(L',R')
delta_p := boundary_merge(P',delta_l,delta_r,x_m)
return (P',delta_p)
```

where merge_by_y merges L' and R' along y axis, and boundary_merge does the following. Inputs are original points P sorted along y, closest distance within L and R, and median point x_m. CLosest distance in P must be eitehr delta_l, delta_r, or a distance between a point in L and R. The two points must lie within delta = min(delta_l, delta_r) of x = x_m, which defines a region in which the points must lie.

```
ALGORITHM boundary_merge(P, delta_l, delta_r, x_m)
delta := min(delta_l, delta_r)
M := {(x,y) \in P | (x >= x_m - delta) and (x <= x_m + delta)}
delta_m := min({min({distance(M[i], M[i+j]) : j \in [1...7]}) : i \in [0...P.size
    ()-7]})
```

3.7.2 Convex hull

1. Quickhull Quickhull does something similar to QuickSort in that it picks a "pivot" element, splits the data around the pivot, and recurses on each split set. Pivot element not guaranteed to split the data into equal sized sets, but is often quite effective.

Take points p in plane and p1, p2 that are known to lie on convex hull (e.g. x, y extrema) and return all points that lie clockwise from p1 to p2. Algorithm removes points that cannot be on hull because they are right of teh line from p1 to p2.

```
ALGORITHM subhull(P, p1, p1)
P' := {p \in P | left_of?(p,(p1,p2))}
if (P'.size() < 2) return [p1] ++ P'
else
    i = max_index({distance(p,(p1,p2)) : p \in P'})
    p_m := P'[i]
    in parallel do
        Hl := subhull(P', p1, pm)
        Hr := subhull(P', p, p2)
    return Hl ++ Hr
```

2. Mergehull

Assumes P is presorted according to x coordinates of points. Hl must be a convex hull on the left and Hr must be a convex hull on the right.

```
ALGORITHM mergehull(P)
if P.size() < 3 return P
else
    in parallel do
        Hl = mergehull(P[0...P.size()/2])
        Hr = mergehull(P[P.size()/2...P.size()])
    return join_hulls(Hl, Hr)
```

Can be improved by modifying the searhc for bridge points so they run in constant depth with linear work. Alternatively, use divide and conquer to separate point set into sqrt(n) regions, solving convex hull on each region recursively, then merging all pairs of those regions using binary search.

3.8 Numerical algorithms

3.8.1 Matrix operations

Matrix multiplication e.g. is highly parallelizable because each loop can be done simultaneously.

```
ALGORITHM matrix_multiply(A,B)
(l,m) := dimensions(A)
(m,n) := dimensions(B)
in parallel for i \in [0...l] do
    in parallel for j \in [0...n] do
        R_ij := sum({A_ik * B_kj : k \in [0...m]})
```

Arguably too parallel – a lot of research focuses on what subset of parallelization is actually needed. Matrix inversion is more difficult to parallelize, but can be done.

3.8.2 Fourier transform

Discrete Fourier Transform often solved using the Fast Fourier Transform algorithm, which is similarly easy to parallelize because of its loops. So, most work has gone into reducing the communication costs (butterfly network topology is called FFT network since the FFT has the same communication pattern as the network.)

```
ALGORITHM fft(A)
n := A.size()
if n == 1 return A
else
    in parallel do
        even := fft({A[2i] : i \in [0...n/2]})
        odd := fft({A[2i + 1] : i \in [0...n/2]})
    return {even[j] + odd[j]e^(2pi * i * j / n) : j \in [0...n/2]} ++ {even[j] -
        odd[j]e^(2pi * i * j / n) : j \in [0...n/2]}
```

3.9 Research issues and summary

Research recently on pattern matching, data structures, sorting, computational geometry, combinatorial optimization, linear algebra, linear and integer programming.

3.10 Paper Review: Parallel Algorithms

3.10.1 Motivation

Introduction to the design and analysis of parallel algorithms: algorithms that specify multiple operations on each step.

3.10.2 Key Ideas

There are several ways of modeling parallel algorithms. These models are necessary because, in the context of having multiple processors and memory modules, the assumption made in sequential models such as random-access machines (RAM) that basic CPU operations (arithmetic, logic, memory access etc.) only require one time step is no longer valid.

The first kind of model is called a multiprocessor model. This is closely related to the underlying hardware of a parallel processor, and takes into consideration the connectivity between processors and memory modules (e.g. is there an interconnection network? If so, what is its topology? Do processors have local memory modules or are they connected to memory through the network?) With the exception of parallel random-access machines (PRAM), a somewhat hypothetical form of parallel processor that assumes one shared memory that each processor can access in one time step, the organization of processors, memory modules, and network determine the cost of each CPU operation.

The second way of modeling parallel algorithms is called work-depth. This is more abstract, and studies an algorithm's potential to be parallelized. Under the work-depth model, W work is considered to be the total cost of the operations an algorithm performs. The maximum length of dependencies between these operations is called the depth D. The ideal parallel time that can be achieved with P processors for a given algorithm is therefeore $T_P \approx W/P + D$ where work is evenly spread out amongst the available processors, but cannot go faster than longest path of interdependent operations.

An algorithm that is designed for ideal conditions using work-depth model can be translated into a more hardware-conscious multiprocessor model in a way that preserves work and does not incur substantial additional slowdowns.

3.10.3 Results

The article describes several algorithms that are common in sequential form in parallel form. Oftentimes, parallelization is used when there are many simple and/or independent operations that can be performed at the same time (e.g. moving pointers in a linked list, or summing adjacent values in a list).

More complex are graph problems. Traditional BFS and DFS can be converted into parallel form but are not necessarily more efficient. Consequently, a variety of techniques such as graph contraction have been developed. The chapter focuses on two such algorithms, a random-mate and deterministic graph contraction. It then expounds on some of the applications of these algorithms to related problems such as finding the minimum spanning tree.

The author also discusses computational geometry and numerical problems. I'm more familiar with these kinds of algorithms so seeing the parallel versions was both interesting and seemed intuitive. I found it interesting that most of the work in the matrix operations world is focused not on finding clever ways of parallelizing, since those algorithms naturally lend themselves to parallel implementaitons, but on optimizing the communication costs i.e. figuring out when it's actually smarter not to parallelize. The little snippet linking the butterfly network architecture in hardware to the fast fourier transform was also quite beautiful.

3.10.4 Novelty

I don't know if anything here was particularly novel but it was certainly educational, and since many of these concepts are novel to me, arguably a success.

Thinking in Parallel

Author(s): [To be filled]

4.1 General Notes

4.2 Paper Review

4.2.1 Motivation

4.2.2 Key Ideas

4.2.3 Results

4.2.4 Novelty

4.2.5 Strengths/Weaknesses

4.2.6 Ideas for Improving Techniques/Evaluation

4.2.7 Open Problems/Directions for Future Work