Algorithms for Hard Problems

- Monte Carlo and Markov Chain Monte Carlo methods,
- backtracking,
- branch & bound and
- alpha-beta pruning.

Why parallel algorithms?

- Monte Carlo and Markov chain Monte Carlo methods are parallelized by performing multiple trials,
- backtracking, branch & bound and alpha-beta pruning: very little interaction required when the search effort is partitioned among several processes.
- Most algorithms are "embarrassingly parallel", however important issues such as dynamic load balancing and termination detection have to be dealt with.

The Monte Carlo Method

- The reliability of a randomized algorithm can be improved, if more random trials are performed.
- Normally only little interaction between different trials is required and parallelization is immediate.
- An example: approximating π.
    - The area of the circle C with radius one equals π, whereas the area of the square $S = [-1, +1]^2$ equals four.
    - If we randomly draw p points from S and if $p(C)$ is the number of points which belong to the circle C, then the ratio $\frac{p(C)}{p}$ converges to $\frac{\pi}{4}$.
    - The more trials, the better the approximation.

An Example: Evaluating Multi-Dimensional Integrals

- Many deterministic integration methods operate by taking a number of samples from a function. However the number of required samples increases with the dimension:
  - A spacing of $\frac{1}{N}$ within the interval $[0, 1]$ requires N points,
  - to obtain a similar spacing of the cube $[0, 1]^k$, N^k grid points are required.
- To approximate the integral $\int_{x \in \Omega} f(x) d^k x$ with Monte Carlo Methods:
  - determine the volume $V(\Omega)$ of the integration region,
  - approximate the expected value $E_\Omega(f)$ of f restricted to $\Omega \subseteq \mathbb{R}^k$ and
  - observe $\int_{x \in \Omega} f(x) d^k x = V(\Omega) \cdot E_\Omega(f)$.
  - How to approximate the expected value? Randomly select points $x_1, \ldots, x_M \in \Omega$ and return the estimate $\frac{\sum_{i=1}^{M} f(x_i)}{M}$.

Markov Chains

A finite Markov chain $\mathcal{M} = (\Omega, P)$ is described by a finite set Ω of states and a matrix P of transition probabilities between states in Ω.

- $P[u, v]$ is the probability to visit $v \in \Omega$, given we currently visit $u \in \Omega$.
- $\sum_{v \in \Omega} P[u, v] = 1$ holds for all states u.

If there is a path with positive probability between any two states in Ω and if $P[x, x] > 0$ holds for all states x:

- $\mathcal{M}$ has a unique stationary distribution π, i.e., $\pi^T \cdot P = \pi^T$ holds. (If $\mathcal{M}$ is in state u with probability $\pi(u)$, then after one step $\mathcal{M}$ is in state v with probability $\sum_{u \in \Omega} \pi(u) P[u, v] = (\pi^T \cdot P)_v = \pi(v)$: $\mathcal{M}$ stays in the stationary distribution π.)
- $\lim_{t \to \infty} P^t[u, v] = \pi(v)$ holds and the frequency with which v is visited does not depend on the starting state u.

Markov Chain Monte Carlo Methods

- Markov chain Monte Carlo (MCMC) methods construct a Markov chain that has a target distribution as its stationary distribution.
- The state of the chain after a sufficiently large number of steps is then used as a sample from the target distribution.
- The major issue: the time to converge against the target distribution may be quite large.

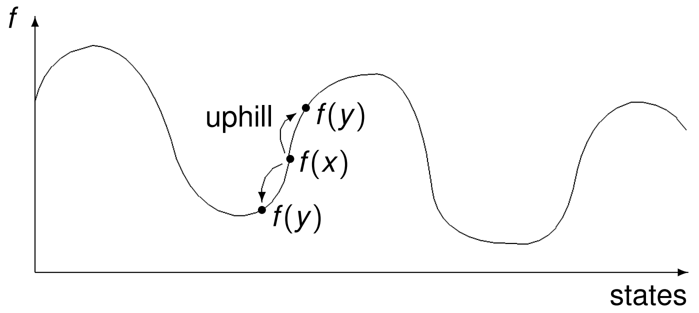
The Metropolis Algorithm

The goal: minimize a function f over some finite domain Ω.

- For any point $x \in \Omega$ let $N(x) \subseteq \Omega$ be the neighborhood of x.
- The Metropolis algorithm starts at some initial point $x \in \Omega$.
  - If the algorithm is currently visiting point x, then it randomly chooses a neighbor $y \in N(x)$.
  - It continues with y, if y is at least as good as x, i.e., $f(y) \leq f(x)$.
  - To escape local minima, an uphill move, i.e., $f(y) > f(x)$, is accepted with probability $e^{-\frac{f(y)-f(x)}{T}}$:

    the larger the "temperature" T, the higher the probability that a bad neighbor is accepted.

The stationary distribution of the Metropolis Algorithm



- Interpret the points in Ω as states of a Markov chain.
  The transition probability from state x to a neighbor y is the probability that y is chosen and accepted.
- The stationary distribution is proportional to $q_T(x) = e^{-\frac{f(x)}{T}}$.
  - The smaller $f(x)$, the higher the probability of x. Great!
  - For hard minimization problems the convergence against the stationary distribution is slow! Not so great.

The Metropolis Algorithm: An Example

In the Vertex Cover problem we are given an undirected graph $G = (V, E)$. Determine a cover, i.e., a subset $C \subseteq V$ of minimal size such that each edge has at least one endpoint in C.

- Define a neighborhood: subsets $U_1, U_2 \subseteq V$ are neighbors iff U_2 results from U_1 after inserting a node or removing a node from U_1.
- Apply the Metropolis algorithm to the empty graph, i.e., $E = \emptyset$. Obviously the empty set is a minimal cover.
    - Assume we start with the cover $x = V$.
    - Initially the Metropolis algorithm removes elements from its current solution.
    - If its current solution x has only few elements, then there are far more larger than smaller neighbors: the Metropolis algorithm begins to add nodes!
- Slowly increase the temperature!

Simulated Annealing

- In physical annealing a material is first heated and atoms can rearrange freely.
  - When slowly cooling down, the movement of atoms is more and more restricted until the material reaches a minimum energy state.
  - A perfect crystal with regular structure corresponds to a global minimum.
- Simulated Annealing:
  - Start with a high temperature T.
  - For any given temperature T: run the Metropolis algorithm sufficiently long and then cool down.
  - For how long do we run Metropolis? Cooling down by how much? Good questions.
- Why Simulated Annealing?
  - Simulated Annealing is applicable without much information on the problem, however the approximation performance may be poor.
  - Parallelization is easy: perform many runs in parallel.

Google's Pagerank

- Google assigns a page rank $\mathrm{pr}(w)$ to a website w via peer review: the more websites with high page rank point to w the higher $\mathrm{pr}(w)$.
- Intention: Take the stationary distribution of the Web Markov chain as page rank.
  - Does the stationary distribution exist? There should be a path with positive probability between any two states.
  - Therefore Google inserts new low-probability links and connects each page w_1 with any other page w_2.
- How to compute the stationary distribution of a transition matrix with several billions of rows and columns?
  - Begin with the uniform distribution π_0 and set $\pi_{t+1}^T = \pi_t^T \cdot P$. Then $\pi_t^T = \pi_0^T P^t$ and $\pi = \lim_{t\to\infty} \pi_0^T \cdot P^t$ is the stationary distribution: $\pi^T \cdot P^t = (\lim_{t\to\infty} \pi_0 \cdot P^t)^T \cdot P = \lim_{t\to\infty} \pi_0^T \cdot P^{t+1} = \lim_{t\to\infty} \pi_0^T \cdot P^t = \pi$.
  - Two facts help: the transition matrix is sparse and convergence against the stationary distribution is fast.
  - Google computes the matrix-vector product $\pi_t^T \cdot P$ in parallel employing several thousand PC's.

Revisiting the Evaluation of Integrals

- Assume we want to determine $\int_{x \in \Omega} f(x) d^k x$ approximately, but the variance of f is large.
- Start an ensemble of “walkers” to move around the integration region randomly in the search of “high-activity” areas.
- A walker checks its current area to determine a point with a considerable contribution towards the integral, respectively to determine the next area to walk into.
- In particular, a Markov chain is constructed for which the integrand “corresponds” to its stationary distribution.

Pseudo Random Number Generators

- A generator G is a deterministic algorithm which, given a seed $x \in \{0,1\}^n$, produces a string $G(x) \in \{0,1\}^{p(n)}$ with $p(n) > n$.
- A statistical test $\mathcal{T}$ is a randomized algorithm which outputs zero or one and runs on inputs of length n in time polynomial in n.
- G passes the test $\mathcal{T}$ if the acceptance probability r_n of $\mathcal{T}$, given a truly random string of length $p(n)$, is not observably different from the acceptance probability g_n of $\mathcal{T}$, given a string $G(x)$.
- G is a crytographically secure pseudo random generator, provided G passes all statistical tests running in polynomial time.

- r_n is not observably different from g_n iff for all $k \in \mathbb{N}$ there is a bound N_k such that $|g_n - r_n| \leq n^{-k}$ for all $n \geq N_k$.
- A generator stretches a random seed into a longer string g_n. To be crytographically secure, G cannot be differentiated from a truly random source within “reasonable” means.

The Blum-Blum-Shub Generator (BBS)

- For a seed s_0 determine the sequence $s_{i+1} = s_i^2 \bmod N$, where $N = p \cdot q$ with primes $p \equiv q \equiv 3 \mod 4$.
- The BBS generator produces the pseudo-random string $G(s_0) = (s_1 \bmod 2, \ldots, s_m \bmod 2)$ with, say, $m = (\lceil \log_2 s_0 \rceil)^k$ for a constant k.
- The BBS generator is cryptographically secure, provided factoring of most numbers $N = p \cdot q$ —with primes $p \equiv q \equiv 3 \mod 4$ — is computationally hard.
- The BBS generator is quite expensive since we have to square in order to get one pseudo random bit.

The Linear Congruential Generator *LC*

- *LC* is defined by its modulus m, its coefficient a and its offset b.
- Generate a sequence x_i of numbers
  - by starting with a seed x_0 and
  - setting $x_{i+1} = a \cdot x_i + b \mod m$.
- The good: *LC* is reasonably fast and has large periods, if m is a sufficiently large prime number.
- The bad: *LC* is not cryptographically secure. It should not be used when high quality pseudo random numbers are required.
- Surpassed in practical applications by the Mersenne Twister.

The Mersenne Twister *MT* 19937

- Its recurrence expands the seed each time by 32 bits and is of the form $x_n = x_{n-227} \oplus (x^U_{n-624} \circ x^L_{n-623}) \cdot A$.
  - All sequence numbers x_n are 32-bit words.
  - x^U_m is the leading bit of x_m and x^L_m is the string consisting of the trailing 31 bits of x_m.
  - A is a 32×32 bit matrix with a 31×31 identity matrix in the upper left. Its last row is 9908B0DF in hexadecimal.
- MT 19937 requires a seed of $19937 = 32 \cdot 623 + 1$ bits, namely 623 strings of length 32 plus the leading bit of x_0.
- The good: *MT* is a very fast generator with the gigantic period length $2^{19937} - 1$. (Its period length is a Mersenne prime, explaining its name.) It is part of the GNU scientific library.
- The bad: it does pass some important statistical tests, but there is no thorough study.

Depth-First Search (DFS)

- DFS is a recursive method. When visiting a node *v* for the first time:
  - mark the node as “visited” and
  - recursively visit all neighbors which are not marked as “visited”.
- DFS visits all nodes in an undirected connected graph. Its advantage is its memory consumption which is bounded by the length of a longest path.
- DFS for graphs is hard to parallelize as we show later, but it is easy for trees.

Decision Problems

In a decision problem we are given a set U of potential solutions and we have to determine whether U contains a true solution.

- For instance let α be a conjunction of disjunctions. U is the set of all truth assignments of the variables of α.
- We have to determine whether U contains a true solution, namely an assignment satisfying α.

The Branching Operator

- Backtracking searches for a solution by trying to construct a true solution step by step from partial solutions.
- It begins with the partial solution $r = U$, which corresponds to the set of all potential solutions.
- Then a branching operator B is applied to r which returns a partition $r_1 \cup \cdots \cup r_k$ of r.
- The branching operator B defines the backtracking tree $\mathcal{T}$:
  - initially $\mathcal{T}$ consists only of the root r. Then we attach children $r_1, \ldots, r_k$ to r to mimic the partition $r = r_1 \cup \cdots \cup r_k$.
  - In general, if B is applied to a node v which is not a singleton set, then we obtain a partition $v = v_1 \cup \cdots \cup v_l$ and correspondingly make $v_1, \ldots, v_l$ children of v.

Backtracking

Backtracking tries to generate only a very small portion $\mathcal{T}^*$ of $\mathcal{T}$.

- Namely, whenever Backtracking finds that a node v does not have a true solution, it disqualifies v. v will not be expanded any further.
- Often backtracking generates $\mathcal{T}^*$ in a depth first search manner:
  - if node v is currently inspected and if v can be disqualified, then backtracking "backs up" and continues with the parent of v.
  - Otherwise backtracking continues recursively with a not yet inspected child of v.
- How to determine whether a node can be disqualified?

Backtracking: An Example

Let $\alpha(x_1, \ldots, x_n)$ be a conjunction of disjunctions.
Determine whether α is satisfiable.

- In our approach partial solutions correspond to partial assignments. To be specific, assume that we already have assigned truth values to all variables x_j for $j \in J$.
  - We determine a disjunction d of minimal size and choose an arbitrary variable x_i appearing in d.
  - The branching operator B then produces two partial solutions by additionally setting $x_i = 0$ respectively $x_i = 1$.
- Why minimal size? To allow for a faster falsification of partial assignments. We run the following test after fixing the value of x_i:
  - we look for any disjunction with exactly one unspecified variable, fix the variable appropriately and continue looking for disjunctions with exactly one unspecified variable.
  - If some disjunction is falsified during this process, then the partial assignment is doomed and we disqualify it.

Parallel Backtracking

- Implement a parallel version of depth-first search for trees.
  - A master process determines the "top portion" T^* of the tree T and distributes the leaves of T^* among the processes.
  - Each process runs DFS for its nodes.
- Implementation issues:
  - What to do when a process runs out of work? We discuss load balancing later.
  - A silly question: How to determine whether all processes are done?
    - ⋆ if a process replies that it is idle, then this process may receive work soon afterwards.
    - ⋆ We discuss the general problem of termination detection later.

Lower Bounds for Minimization Problems

Our goal is to minimize a function f over a finite domain Ω.

- Branch & Bound again utilizes the branching operator B, but does not differentiate between potential and true solutions: we look for a cheapest solution.
- As for backtracking the branching operator defines a tree $\mathcal{T}$.
- The crucial requirement of Branch & Bound is the existence of a lower bound α with

$$\alpha(v) \leq \min\{f(x) \mid x \in \Omega \text{ is a leaf in the subtree of } \mathcal{T} \text{ with root } v\}.$$

The Lower Bound: An Example

In the traveling salesman problem (TSP) we are given a set of nodes in the plane and are asked to compute a path of shortest length traversing all nodes.

- A somewhat related, but computationally far easier problem is the minimum spanning tree problem (MST):
  - Given is an undirected graph G with weighted edges.
  - Determine a subtree T of G such that its sum of edge weights is minimal.
- What is the relation between TSP and MST?
  - If a path P of length L traverses all nodes, then we have found a spanning tree of weight L, namely the path P.
  - We have found a lower bound for all possible TSP-paths and the lower bound is computable within reasonable resources.

Branch & Bound: The Algorithm

- Branch & Bound begins by constructing a "good" initial solution x with the help of a heuristic and sets $\beta = f(x)$.
- Initially only the root of $\mathcal{T}$ is unexplored.
- In its general step Branch & Bound has computed a set U of unexplored nodes and it chooses an unexplored node $v \in U$ to explore.
  - If $\alpha(v) \geq \beta$, then no solution in the subtree of v is any better than the best solution found so far. Branch & Bound disqualifies v.
  - If v is a leaf corresponding to a solution x, then β is updated by $\beta = \min\{\beta, f(x)\}$.
  - If v is not a leaf, then all children of v are generated and added to the set U of unexplored nodes.

Parallel Branch & Bound

- A master process determines the top portion of the branch & bound tree and communicates it to the remaining processes.
- Each process *i* works on its subproblem by representing its set U_i of unexplored nodes by its own private priority queue.
- So far no communication is required.
  - If a process runs out of work, then apply load balancing schemes as for backtracking.
  - Each process broadcasts a better upper bound immediately.
  - To obtain good upper bounds as fast as possible, some parallel implementations let processes also exchange promising unexplored nodes.

Playing Games

- Two players Alice and Bob play a game.
- Alice begins and the two players alternate.
- The game ends after finitely many moves with a payment to Alice: for instance with payments $-1, 0$ or 1
  - Alice wins, if she receives a payment of 1 and
  - Bob wins, if Alice receives a payment of -1.
- Determine a strategy for Alice that guarantees her the highest possible payment.

Any such game has a game tree $\mathcal{B}$.

- Its root r corresponds to the initial configuration and is labeled with Alice.
- For any node v of $\mathcal{B}$ and for any possible move in v: generate a child w of v and label it with the opposing player.
- If the game is decided in v, then v becomes a leaf and we label v with the payment $A(v)$ to Alice.

Alpha-Beta Pruning: The Idea

Assume that we traverse $\mathcal{B}$ in a depth-first manner and that we reached a node *v* belonging to Alice.

- Let *u* be an ancestor of *v* belonging to Bob and assume that Bob can restrict Alice to payments of at most β, when reaching *u*.
- If Alice can enforce, for some child *w* of *v*, a payment of at least $\alpha \geq \beta$, then Bob does not profit form reaching *v*. Moreover Bob can prevent Alice from reaching *v*.

  The evaluation of *v* can be stopped!
- The invariant: for any node *v* work with two parameters α and β.
  - α is the highest score for Alice detected so far for an ancestor of *v* belonging to Alice.
  - β is the lowest score for Alice detected so far for an ancestor of *v* belonging to Bob.

Alpha-Beta(v, α, β) I

The first call involves the root with $\alpha = -\infty, \beta = +\infty$.

(1) If v is a max-leaf, then return $\alpha = \max\{\alpha, A(v)\}$.
    If v is a min-leaf, then return $\beta = \min\{\beta, A(v)\}$.
    // We make sure that the invariant holds for v.

(2) Otherwise work recursively.

Alpha-Beta(v, α, β) II

- If v is a max-node, then // Alice makes a move.
  - Max $= \alpha$,
  - traverse all children w of v: if alpha-beta$(w, \alpha, \beta) \geq \beta$, then stop the traversal and return α.
    // Bob can prevent Alice from reaching v .
    Otherwise Max $= \max\{$Max, alpha-beta$(w, \alpha, \beta)\}$.
    // Alice makes her best move.
  - Return $\alpha =$ Max.
- If v is a min-node, then // Bob makes a move.
  - Min $= \beta$,
  - traverse all children w of v: if $\alpha \geq$ alpha-beta(w, α, β), then stop the traversal and return β.
    // Alice can prevent Bob from reaching v .
    Otherwise Min $= \min\{$Min, alpha-beta$(w, \alpha, \beta)\}$.
    // Bob makes his best move. //
  - Return $\beta =$ Min.

Properties of Alpha-Beta Pruning

For the subtree with root *v*, let *A* be the largest payment reachable by Alice. Assume that α and β are obtained before visiting *v*.

(a) If *v* belongs to Alice, then $\max\{\alpha, A\}$ is returned, provided $A \leq \beta$.

(b) If *v* belongs to Bob, then $\min\{\beta, A\}$ is returned, provided $\alpha \leq A$.

Let $\mathcal{B}$ be a complete *b*-ary game tree of depth *d*.

- There is an evaluation of $\mathcal{B}$ by alpha-beta which inspects at most $\text{opt} = b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$ nodes.
- In the best case, alpha-beta reduces the search effort from $\Theta(b^d)$ to $\Theta(\sqrt{b^d})$: in comparison with a brute force evaluation the number of simulated moves is doubled.
- The best case occurs in practical applications, if depth-first search uses a good heuristic to pick the next move.

Parallel Alpha-Beta Pruning

We have to find a trade-off between the search overhead (the increase in the number of inspected nodes in comparison with a sequential implementation) and the communication overhead.

- Assume the game tree $\mathcal{G}$ is the complete b-ary tree of depth d.
- If a parallel alpha-beta pruning implementation with $p = b$ evaluates all children of the root in parallel, then each process inspects $\mathrm{opt}_{d-1} = b^{\lceil (d-1)/2 \rceil} + b^{\lfloor (d-1)/2 \rfloor} - 1$ nodes in its subtree.
- If d is even, then $\mathrm{opt}_{d-1} = b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor}/b - 1 \geq b^{\lceil d/2 \rceil} \geq \mathrm{opt}_d/2$ and the best achievable speedup is two!
- The search overhead is $p \cdot \mathrm{opt}_{d-1} - \mathrm{opt}_d \geq p \cdot \mathrm{opt}_d/2 - \mathrm{opt}_d = (p/2 - 1) \cdot \mathrm{opt}_d$.

Young Brothers Wait Concept (YBWC)

How to decrease the search overhead?

- Evaluate the leftmost child (the eldest brother) before processes work on the remaining siblings (the younger brothers).
- Even if the leftmost child is not optimal, its (α, β) value may help to narrow the search windows for its siblings.
- When good moves are explored first,
  - it pays to throw all computing power at the subtree of the leftmost child
  - and then to process siblings in parallel.
- We describe two parallel implementations based on YBWC.

Partial Synchronization

- A leftmost child v is a synchronization node, whenever a parallel implementation enforces YBWC by exploring v before its siblings.
- In many implementations all nodes of the leftmost path $\mathcal{P}$ of $\mathcal{G}$ are synchronization nodes.
  - Only one process is at work when the deepest node of $\mathcal{P}$ is evaluated and more processes enter only after higher nodes of $\mathcal{P}$ are reached.
  - YBWC keeps the search overhead low at the expense of unbalanced work loads and a higher communication overhead.
- If the computation progresses, there is sufficient work and load balancing becomes an important issue:
  - Idle processes send work requests.
  - If a process q receives a request from process p, it checks its current depth-first path p_d and chooses a sibling s of a node of p_d. It sends s to p and enters a master-slave relationship with slave p.
  - The slave p may become a master after receiving a work request.

Asynchronous parallel hierarchical iterative deepening (APHID)

- APHID uses fixed master-slave relationships. The master explores the top levels, assigns “leaves” to slaves and continuously repeats his evaluations of the top levels:
  - accepting updates from the slaves,
  - performing heuristic evaluations of “open” leaves,
  - informing slaves to terminate a task,
  - performing load balancing by reallocating tasks from overworked to moderately busy processes,
  - and informing a slave about the (changed) relevance of its leaves. (The relevance of a leaf is determined by YBWC and the search depth achieved so far for the leaf: the smaller the search depth the higher the priority, thus allowing the leaf to catch up.)
- Instead of a single master, a master-slave hierarchy may be used. Thus the communication overhead should shrink.
- APHID has a considerable search overhead. However the masters assign higher relevance to leftmost leaves.