Storing arrays in memory

Representing integers with a fixed number of bits

- What was the meaning of the negative result in the previous example?
- Suppose that you have b=4 bits to represent integers. You then can represent at most $2^b=16$ different numbers. You want both negative and positive numbers. The convention in this case is to go from $-2^{b-1}=-8$ to $2^{b-1}-1=7$, using the first $2^{b-1}=8$ entries for positive numbers and the remaining half for negative ones.
- If you "go over" the maximum or "go below" the minimum (i.e. if you *overflow*), the numbers cycle around. So in out 4-bits example, you get bizarre results like `7+1 → -8`
- With b=64, this happens rarely, but it still sometimes does. It's important to know about it. (YouTube knows something about it...)
- There also exist "unsigned" types. In that case you have all values from 0 to 2^b-1, but no negative numbers. In the 4-bits case, you would then have results like `15+1 → 0`, `5*4 → 4` etc. (everything is computed "`mod 16`").
- NOTE: this hypothetical `int4` type does not exist, the smallest one is `int8` (ranges from `-128` to 127). `uint8` is common in images data (ranges from 0 to 255)

bits	int4	uint4
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

(`int4` and `uint4` don't really exist)

```
np.zeros(tuple)
np.ones(tuple)
np.full(tuple, x)

np.arange(...)

np.linspace(start, end, n)

a.ndim
a.shape
a.size
a.fill(x)

a.max()
a.min()
a.sum()
a.mean()
a.argmax()
a.cumsum()
...
```

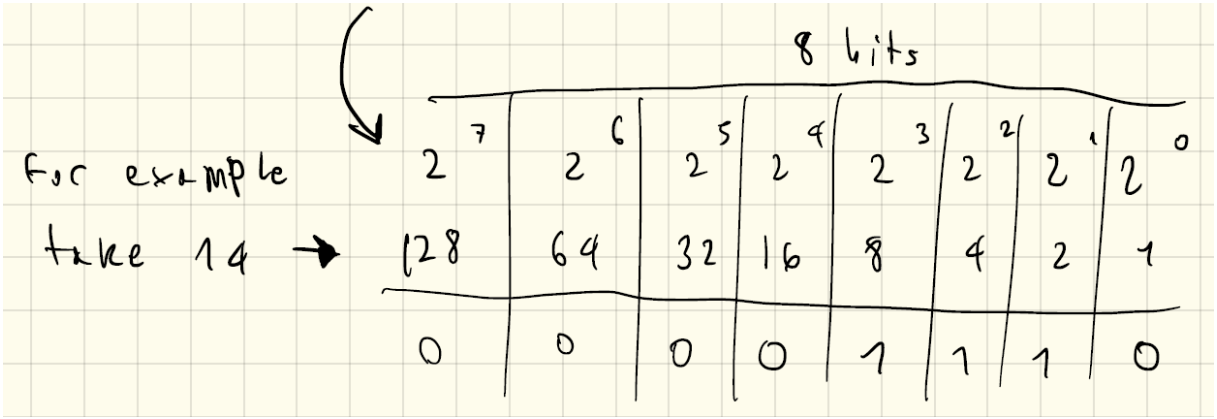
RAM is a linear container

1 byte = 8 bits

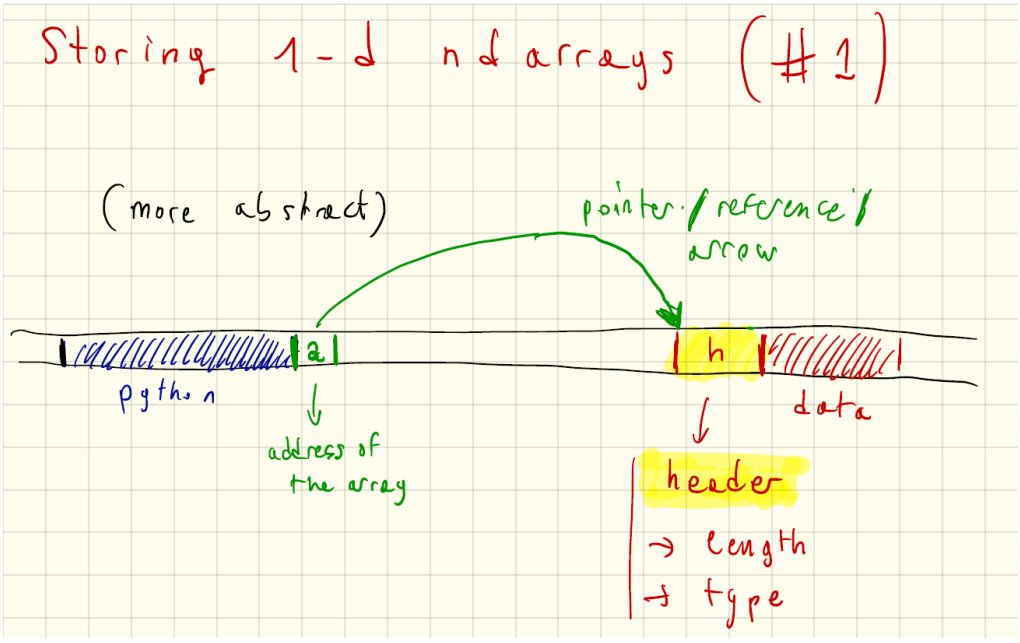
Adress stored in 1 byte (10^9 bytes is in 1 GB), there are billions of addresses.

Bits have the state 0 or 1 depending on the past computations/ random initialization.

Binary representation of numbers in terms of 8 bits (0 and 1). We write numbers as a sum of powers of 2, from 0^{th} power to 7^{th} power and the corresponding bit reflects the coefficient in front of the elements of the sum.

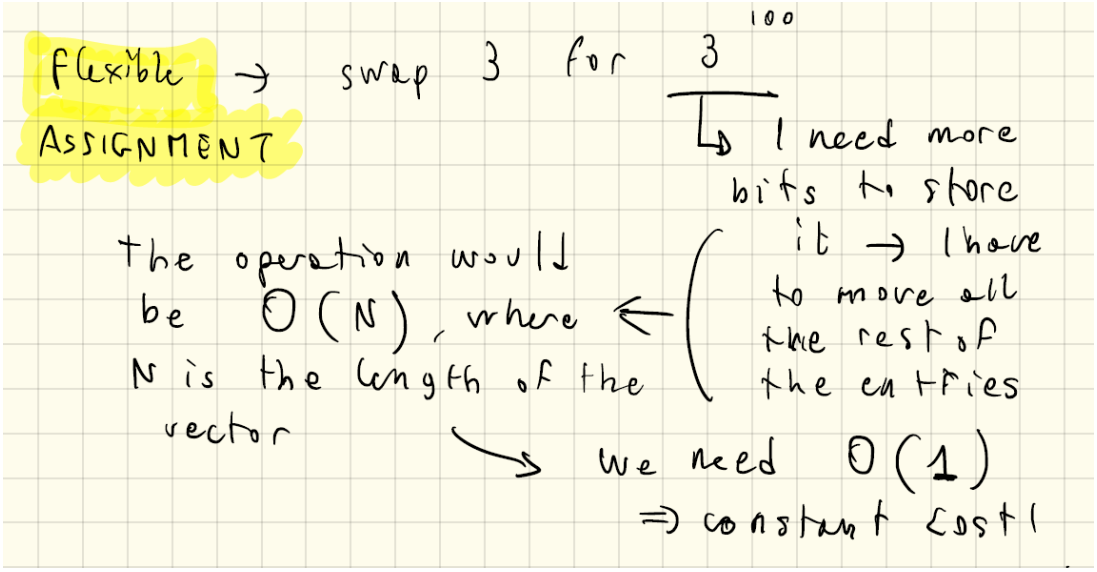


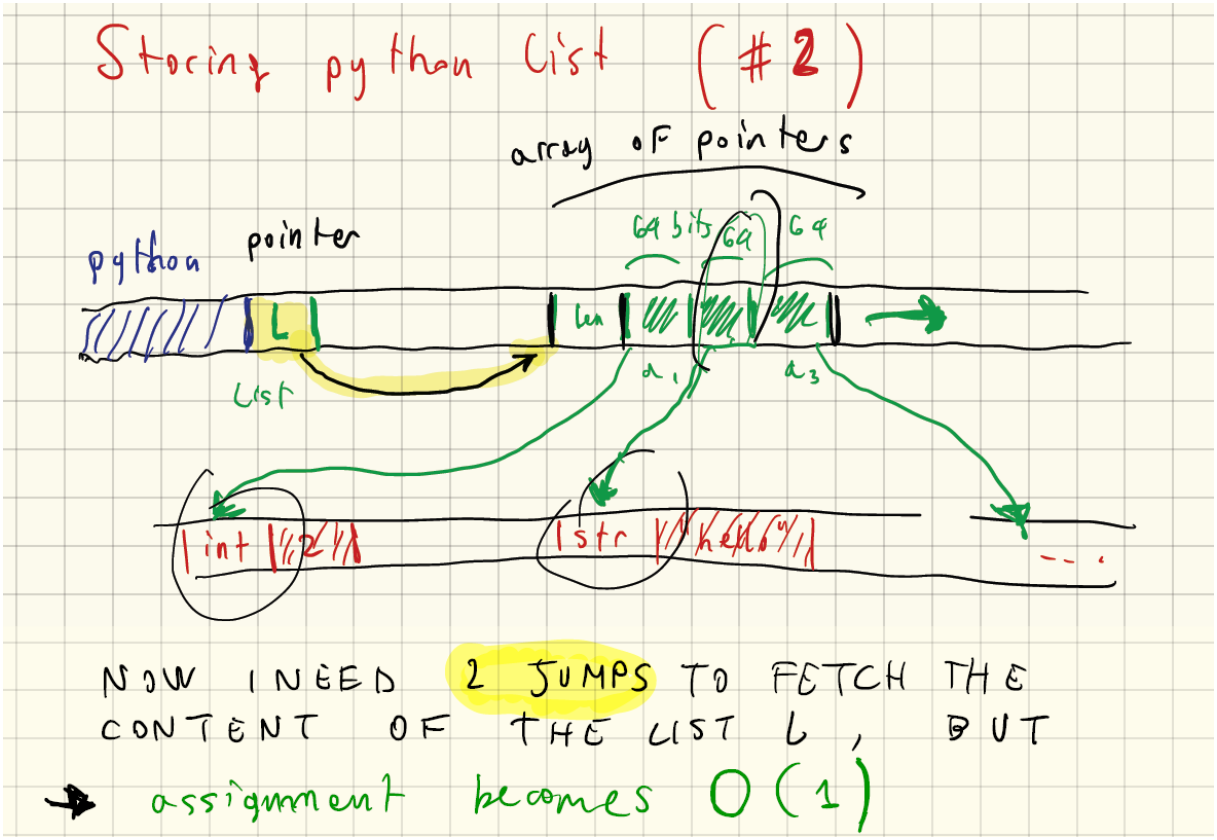
To store an array of numbers, use consecutive slots in RAM.

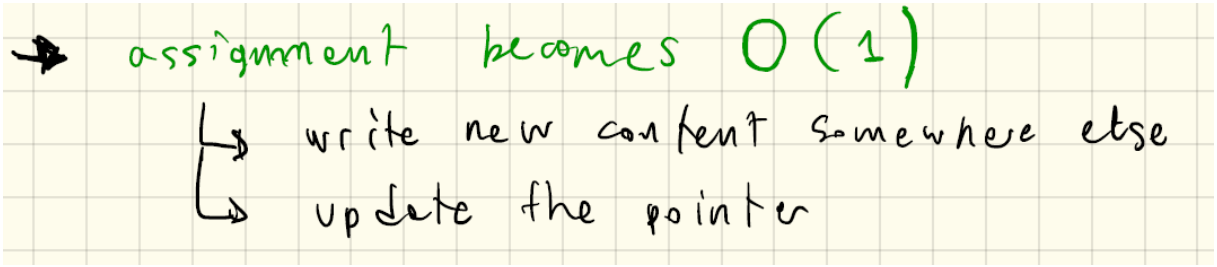


Header contains information about the length and the type of the array of data (entries of the ndarray) which follows it. It allows for efficient storing of arrays in memory and operations involving data of the same type are efficient in letting the computer avoid type conversions.

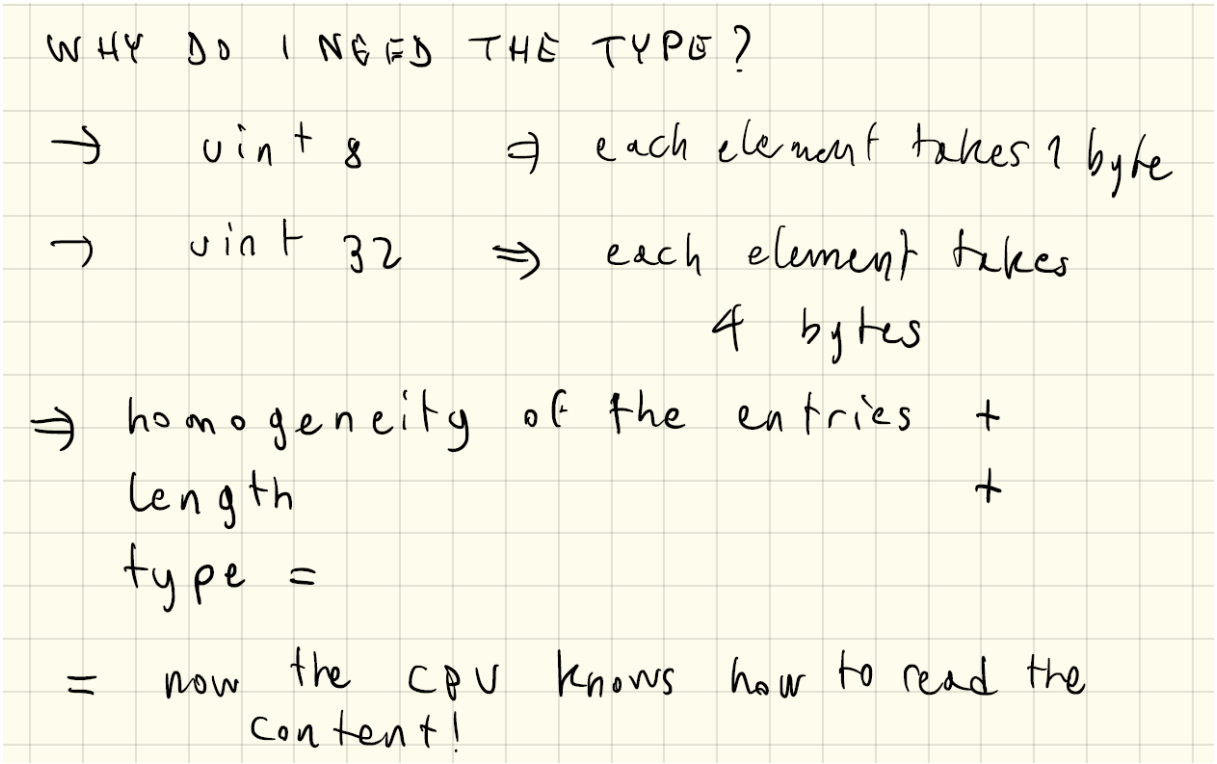
Feature of lists:





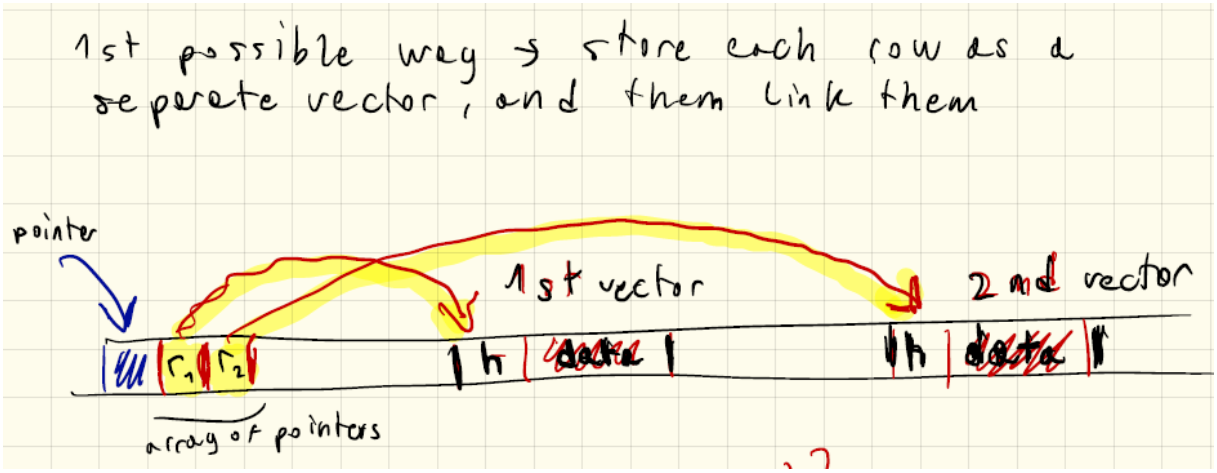


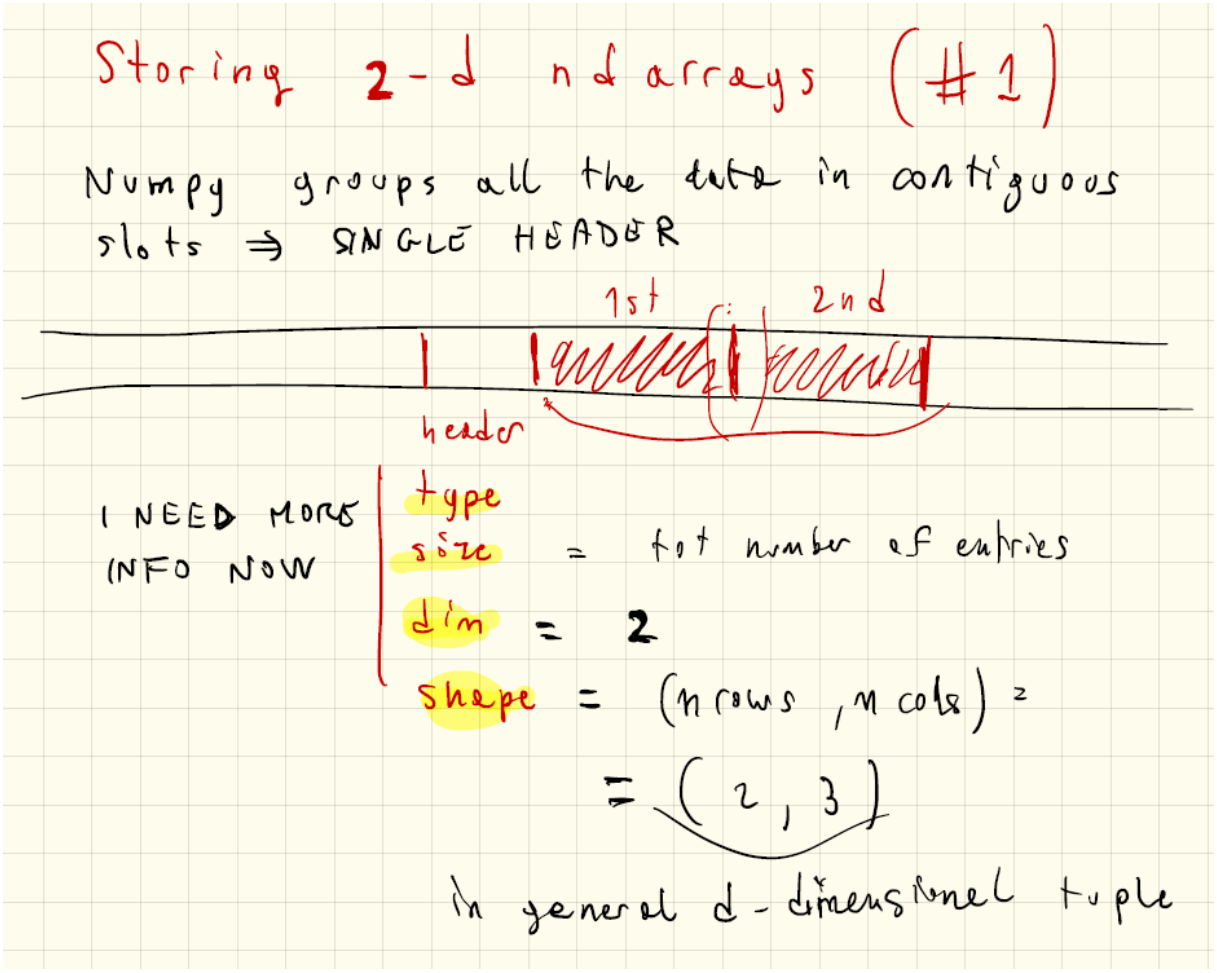
Pointer from Python points at the header which contains information about the length of the array of pointers which follows it. The pointers from this array point at headers of individual entries of the list (somewhere else in memory), which just contain information about datatype of those entries. So there are two pointers and two headers, information about the length of the list and datatypes of particular elements is distributed between header of the array of pointers and headers corresponding to each entry.



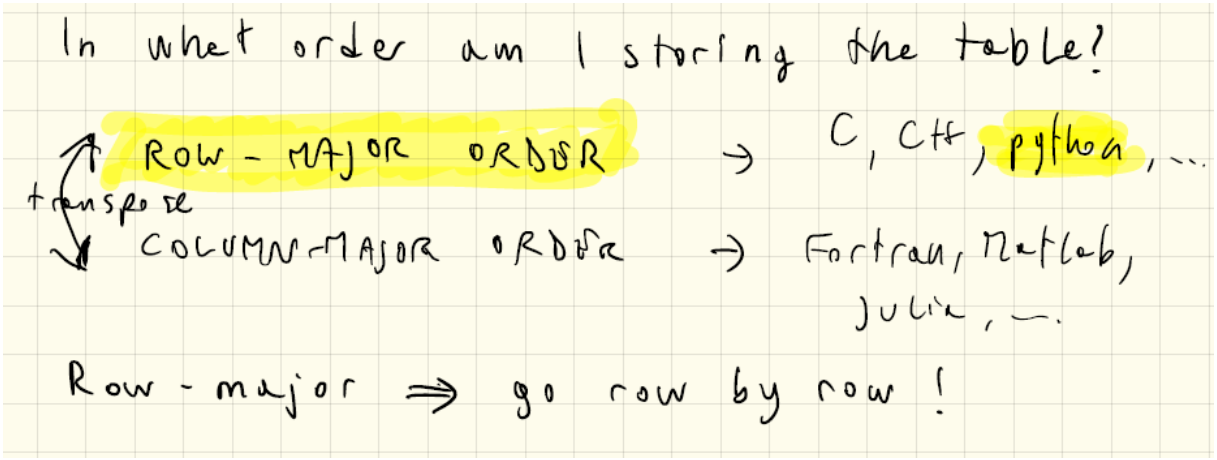
In the case of ndarrays, the type of entries is homogenous so given the information contained in the header about the length of an array and datatype (which reveals information how many bytes are taken by 1 element), CPU knows how to read the content.

Storing 2-d lists: 2nd layer of pointers (so now there are 3 layers). Similar to the following way:





All info contained in the header that allows for such storing.



Optimization and greedy approaches

Optimization problem – definition: given a function c : X → R, where X is any set, find the value x in X such that c(x) is minimum. Possible constraints: X may be a subset of R^d, parameters could have a limited range and interactions.

A discrete optimization problem is a problem in which the space of parameters X is discrete (and possibly finite).

The number of all possible paths grows exponentially with the size of the graph.

Greedy approaches

Let's say that we have at least some notion of "neighborhood" within our X, so that given two elements x and y we can tell whether they are "close" or not.

- Pick an initial guess $x^{(0)}$ in $\mathcal{X}$ (somehow, maybe at random)
- Repeat for all time steps t:
  - Pick a neighbor y of $x^{(t)}$ (somehow, maybe at random)
  - If $c(y) \leq c\left(x^{(t)}\right)$, then $x^{(t+1)} = y$, otherwise $x^{(t+1)} = x^{(t)}$
- Keep going until you found an x such that all its neighbors are worse. This is a **"local minimum"**

Greedy is susceptible to local minimum: multiple attempts (restarts) can help.

- Represent the configuration x internally in the memory (choose a representation)
- Compute the cost (could be expensive to do, not a big problem itself)
- Pick an initial configuration at random
- Pick a random neighbor for any given configuration (propose a move)
- Decide whether the new configuration is better ("delta-cost")

We're going to implement this!

In general we'd like that "local" moves imply O(1) computational effort in updating the cost.

PARTIAL-BUILDUP GREEDY APPROACH: applicable when the elements of X are made of "pieces" and you can define your cost function c on just a part of x. Then you can build your proposed solution one piece at a time; every time you choose the option that gives you the minimum cost so far. Graph example: pick the shortest outgoing edge from A, then the shortest from there (as long as it doesn't go back to a previous node), etc. Maybe you'll get to B?

The Travelling Salesman Problem (TSP)

- Celebrated NP-C problem (no efficient way to reach the optimum in the worst-case)
- Given: n cities (nodes), and the distance d_{ij} between any two cities i and j (symmetric)
- Find the shortest route that goes through every city exactly once and goes back to the starting point ("shortest Hamiltonian path on a graph")
- $\mathcal{X}$ = space of all possible Hamiltonian paths
- $c(x)$ = total distance along the path
- $|\mathcal{X}| = n!/(2n)$ = num. of ways to permute the cities ($n!$), ignoring differences in the starting point (divide by n) and the direction (divide by 2)

Greedy approaches to TSP:

- Partial-buildup greedy approach: start from a city at random, choose each new city as the closest to the last chosen one, excluding the cities already visited. Proceed until all cities are chosen, at that point go back to the first.
- Random-search greedy: route represented by a permutation of cities' indices and close routes (differing by a local move which can follow for example "swap-paths" rule, picking two edges i→j and l→k and swapping to i→l and j→k). Only 4 edges involved in delta-cost computation.

Markov Chain Monte Carlo

Monte Carlo simulations

Estimating pi, areas and volumes with Monte Carlo.

Area (2-d): estimate the area of a quarter circle, throw n points uniformly in a unit square ([0, 1] x [0, 1]), for each point check if $x^2+y^2<=1$, then area of a quarter circle=(points inside/total points)*area of the square (pi=4*=(points inside/total points) for unit square).

General case: estimate the volume of m-dimensional region. Throw n points uniformly in a bounding box of known volume, for each point check if it's inside the region (for unitary volume it is $x_1^2+x_2^2+\ldots+x_m^2<=1$, estimate the volume: (ratio of points)*volume of the box (often 1).

Additional remarks: 1. Error decreases like $\frac{1}{\sqrt{n}}$ for any given m, but using a grid (Riemann-style) would decrease much more slowly $\frac{1}{m\sqrt{n}}$. 2. Requirements: tight enough bounding box so there is a good chance of hitting the region and fast check of inclusion in the region.

Generalization to integrals: Monte Carlo integration (especially useful for higher-dimensional integrals). Estimating the value of a multidimensional integral (m-dimensional problem) by throwing

n points uniformly in a bounding (m-dimensional) box of known volume and weighing each point with the value of the function to estimate the definite integral (evaluating integrand at randomly chosen points):

$$\left(\frac{1}{n}\sum_{i=1}^{n} f\left(\vec{x}_i\right)\right) \times (\text{volume of the box})$$

Issues: if the function is almost 0 almost everywhere except in tiny regions where it's large. Then it may be hard to hit those regions by chance. If the function is smooth enough, we may find them.

Stochastic processes

A stochastic process is a sequence of random variables X^0, X^1, X^2, …, assuming that each of them produces values from the same space X. The sequence may be finite or infinite. The space X may be continuous or discrete, and it's often multidimensional. We'll focus on the discrete case.

A realization of the stochastic process is a sequence of values x^0, x^1, x^2,… each of which belongs to X. This is called a trajectory, because we think of the index of each value as representing a time. You may think of a point moving inside X, in (random) steps.

We are normally interested in studying how the probability distribution of one of the variables at time , , depends on what happened at the previous times (given the trajectory x^0, x^1, …, x^{t-1} up to time t-1.

$$P\left(X^t = x^t \mid X^0 = x^0, \ldots, X^{t-1} = x^{t-1}\right)$$

We use stochastic processes to model sequences of events that happen at random, but that could be influenced but what happened in the past.

Forgetful processes – Markov chains

Thus the simplest "interesting" case is when each new event only depends on the previous one, and all of the more distant past is "forgotten" (ignored). Markov property (simplified notation):

$$P\left(x^{t+1} \mid x^t, x^{t-1}, \ldots, x^0\right) = P\left(x^{t+1} \mid x^t\right)$$

The probability for the whole trajectory (up to some) can thus be decomposed as a chain:

$$\begin{aligned} P\left(x^0, x^1, \ldots, x^t\right) &= P\left(x^t \mid x^{t-1}\right) \ldots P\left(x^2 \mid x^1\right) P\left(x^1 \mid x^0\right) P\left(x^0\right) \\ &= \left(\prod_{t'=0}^{t-1} P\left(x^{t'+1} \mid x^{t'}\right)\right) P\left(x^0\right) \end{aligned}$$

Every step is taken independently from the configuration that we reached, but we are also constrained by the current configuration because we can only make local moves, hence conditional probabilities.

The probability of ending up in x^t at time t can be written as (summing over all possible previous states):

$$P\left(x^t\right) = \sum_{x^{t-1}} P\left(x^t \mid x^{t-1}\right) P\left(x^{t-1}\right)$$

Transition probability

Marginal probabilities

"Incoming flow" into a state. Each state has a "mass" at time which gets multiplied by a "transition rate", and they all get summed up. As t→infinity, these masses stabilize, converge and from then on they do not vary between different points in time.

Random walk on a grid

Suppose you have an infinite regular square grid on a plane. The collection of all possible states X is represented by all possible pairs of integer numbers (i, j).

If we start at (0, 0), then:

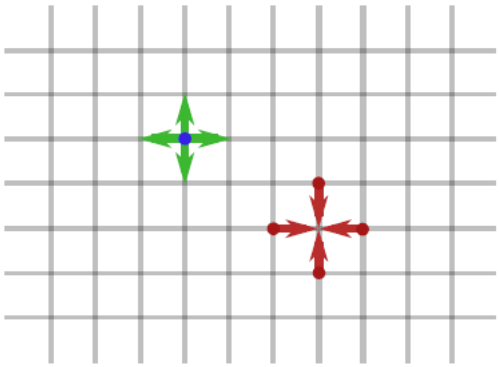
$$P\left(X^0 = (i,j)\right) = \begin{cases} 1 & \text{if } (i,j) = (0,0) \\ 0 & \text{otherwise} \end{cases}$$

For infinite grid (with no edge nor corner points):

$$P\left(X^{t+1} = (i,j) \mid X^t = (k,l)\right) = \begin{cases} 1/4 & \text{if } (i = k \pm 1 \text{ and } j = l) \text{ or } (i = k \text{ and } j = l \pm 1) \\ 0 & \text{otherwise} \end{cases}$$

Probability update formula:

$$\begin{aligned} P\left(X^{t+1} = (i,j)\right) \quad = \quad & \frac{1}{4} P\left(X^t = (i+1,j)\right) + \\ & \frac{1}{4} P\left(X^t = (i-1,j)\right) \\ & \frac{1}{4} P\left(X^t = (i,j+1)\right) \\ & \frac{1}{4} P\left(X^t = (i,j-1)\right) \end{aligned}$$



Current position is at time t and by arriving at that position at time t from some another position at time t-1, we update the probability of the state at time t, and possible moves going out of the state at t could go in 4 directions at t+1. In general probability update happens based on the previous step, it is incoming, inflow while trajectory is about outgoing move from the current state.

Here we sample 1000 trajectories in parallel and take snapshots at different times, and it starts looking like the (marginal) probability distribution $P(x^t)$.

Markov chains probability evolution: linear algebra

Say that X is discrete and finite. Say that it has M elements. Say that you order them somehow, thus you have a label for each element: 1, 2,…, M (for example a finite grid with M possible states).

- Then you can encode the transition rates $P\left(x^{t+1} \mid x^t\right)$ in a $M \times M$ matrix. For each "source" state i and "destination" state j and time t, you have a probability of getting to j at $t+1$ assuming that you are in i at time t. Call this matrix $Q_{ji}^{(t)}$.

ROWS: DESTINATIONS (even if denoted by j), COLUMNS: SOURCES.

- You can also encode the marginal probabilities $P\left(x^t\right)$ in a vector of size M, call it $\vec{\rho}^{(t)}$

DO NOT CONFUSE LABELS FOR CONFIGURATIONS WITH ACTUAL CONFIGURATIONS![1]

[1] Labels allow us to encode transition probabilities between states in a transition matrix and marginal probabilities in a vector. You have to uniquely relate coordinates to labels to be able to work.

Notational gotchas here!

- $x^t, x^{t+1}, \cdots \in \mathcal{X}$ denote configurations. E.g. in the random walk case they could be $(0,0)$ or $(-3,6)$
- $i, j, \cdots \in \{1, 2, \ldots, M\}$ are configuration labels. Think of a random walk on a finite $L \times L$ grid, then start labeling row by row
- in $Q_{ji}^{(t)}$, i is the source, j is the destination

The formula for the evolution of the marginals in a Markov chain can be expressed in linear algebra notation:

$$P\left(x^{t+1}\right) = \sum_{x^t} P\left(x^{t+1} \mid x^t\right) P\left(x^t\right) \longrightarrow \rho_j^{(t+1)} = \sum_{i=1}^{M} Q_{ji}^{(t)} \rho_i^{(t)} \longrightarrow \boxed{\vec{\rho}^{(t+1)} = Q^{(t)} \vec{\rho}^{(t)}}$$

Marginal probability of some state j at time t+1 is the sum of products of marginal probabilities of previous states at time t times the corresponding transition rates (this can be represented by the transition probability from state i to state j given by Q[ji] – first, row index denotes the state to which we are transitioning and the second, column index, the state from which, i.e. which we are leaving times the marginal probability of state i at time t, SUMMED OVER ALL POSSIBLE VALUES OF I → this is equivalent to the product of j^{th} row of Q with vector $p^{(t)}$). So we can generalize and write the vector of all marginal probabilities at time t+1 as a product of transition matrix and marginal probability vector from time t.

If the transition rates do not depend on time ($Q^{(t)}$=Q) as for a random walk, then:

$$\vec{\rho}^{(t)} = Q\vec{\rho}^{(t-1)} = Q\left(Q\vec{\rho}^{(t-2)}\right) = \cdots = Q\left(Q\left(Q\ldots\left(Q\vec{\rho}^{(0)}\right)\right)\right)$$
$$= \left(QQQ\ldots Q\right)\vec{\rho}^{(0)} = Q^t\vec{\rho}^{(0)}$$

Stochasticity property and stochastic matrix

From any given initial state, we must end up somewhere (transition probabilities from any given source x sum up to 1), which has implications on the matrix representation of a stochastic process. Sum of elements in each column must be 1 and entries are non-negative. Diagonal elements (probability of staying, i.e. transition to the same state) are determined by all other elements (probability of moving) (condition is that the columns must be normalized).

$$\sum_{x'} P\left(x' \mid x\right) = 1 \qquad \forall x \in \mathcal{X}$$

$$\sum_{j=1}^{M} Q_{ji} = 1 \quad \forall i \in \{1, \ldots, M\}$$

$$Q_{ii} = 1 - \sum_{j \neq i} Q_{ji} \quad \forall i \in \{1, \ldots, M\}$$

Stationary distribution

A stationary distribution of a Markov chain is a probability distribution that remains unchanged in the Markov chain as time progresses. Typically, it is represented as a row vector π whose entries are probabilities summing to 1, and given transition matrix P, it satisfies. π=πP.

Eigenvalue and eigenvectors. Vectors describing a stationary distribution (that need not to be unique) are eigenvectors of the transition matrix whose eigenvalue is unity. A Markov matrix A always has an eigenvalue 1. All other eigenvalues are in absolute value smaller or equal to 1. So, why is 1 an eigenvalue of a Markov matrix? Because every column of A — I adds to 1 — 1 = 0. So, the rows of A — I add up to the zero row, and those rows are linearly dependent (according to definition if they add up to 0 with not all zero coefficients), so A — I is singular. So, A is an eigenvalue of A. We can compute this limit distribution (stationary distribution for a particular case of markov chain with transition matrix q) by computing eigenvectors of q. **The vector whose components represent probabilities of respective states given by the stationary distribution is an eigenvector of the transition matrix Q corresponding to the eigenvalue 1.**

$$\vec{\rho}^{(t)} = Q^t \vec{\rho}^{(0)}$$

As t tends to infinity, the trajectory will just keep going jumping randomly between the states. Probability however will converge to a limit distribution called a stationary distribution.

$$\vec{\rho}^{(t)} \xrightarrow[t \to \infty]{} \vec{\rho}^{\star}$$

This stationary distribution $p^{(t)}$, to which p converges as t tends to infinity is characterized by the property:

$$\vec{\rho}^{\star} = Q\vec{\rho}^{\star}$$

Doesn't change with time.
In linear algebra terms: it's an eigenvector with eigenvalue 1.

In rough practical terms: run a simulation "long enough", and then when you look at which state you find it in, you can think of it as having been sampled from the stationary probability distribution.

Run a stochastic process until it converges to stationary distribution (or more easily, you can basically start with the eigenvector of Q corresponding to the eigenvalue 1 as the initial probability distribution) and then as you converged (or otherwise reached) stationary distribution, you have a guarantee you will stay there, i.e. your states will be distributed according to stationary distribution

for all successive repetitions; you can run the simulation for the number of times corresponding to the desired sample size and sample the states at which the process is found at a given point in time. This way you sample from the stationary distribution. (You can create a vector p_est, whose component i is p_est_i=n of times state i is reached in the simulation/total number of runs of the simulation (which is the sample size). So ofc if your sample size is large enough and representative of the population, you would be able to approximate the vector describing the distribution from which we sample by p_est – it is quite obvious statistical fact).

If we have a target distribution from which we want to sample, we want to be able to create a transition matrix Q which is both a stochastic matrix and whose eigenvector corresponding to its eigenvalue 1 is the vector representing the target distribution. You can construct such a matrix using Metropolis-Hastings rule.

Reconstructing the Markov chain from a given stationary distribution

Also it is hard to analytically find a stochastic matrix whose eigenvector corresponding to the eigenvalue 1 is given?

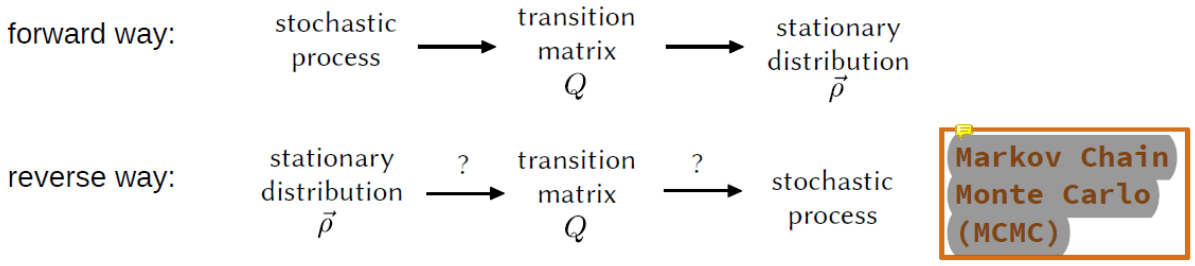
In order to sample from some generic probability distribution (which does not follow a common model), whose normalizing constant is unknown (thus it is complicated to sample directly, we do not know the exact distribution, even the function behind it), we want to build a Markov chain (which entails finding a transition matrix Q) such that its stationary distribution is the probability distribution that we want to sample from. We want to run a stochastic process for a long time (both in order for it to converge to the stationary distribution and to be able to gather big enough sample) and obtain samples from it. We can do that provided that the process is ergodic.

Ergodic: A Markov chain is called an ergodic chain if it is possible to go from every state to every state (not necessarily in one move), connectedness. In practice this means that statistical sampling can be performed at one instant across a group of identical processes or sampled over time on a single process with no change in the measured result. Otherwise, if there was for example a break in the grid, the starting position will affect the reachable states and some will be 0, while for another run they could be non-negative and states which previously had non-zero probability can now have a zero probability.

Extra: In probability theory, a stationary ergodic process is a stochastic process which exhibits both stationarity and ergodicity. In essence this implies that the random process will not change its statistical properties with time and that its statistical properties (such as the theoretical mean and variance of the process) can be deduced from a single, sufficiently long sample (realization) of the process.

Basically we can either sample from multiple identical stochastic processes or draw multiple samples from one (of course in both cases until enough time has passed for the process to reach its stationary distribution).

There are in general infinite processes that realize any given stationary distribution. Moreover, there is always one family of processes that can be built in a systematic way using the Metropolis-Hastings algorithm.



It is called Markov Chain Monte Carlo (MCMC) because to sample from some target distribution p, we construct a Markov Chain and then we sample randomly (this random sampling also allows us to reconstruct the target distribution, as sample gets larger the sample distribution approaches the population distribution and here the population is assumed to follow the target distribution).

Metropolis Hastings algorithm

First we find proposal and acceptance matrix given the target distribution p. Then we find the transition matrix Q given the proposal and acceptance matrices.

MCMC method. To find a stochastic process (belonging to a particular family out of many possible stochastic processes which could satisfy our requirements) that converges to the probability distribution we want to sample from (whose stationary distribution is our target distribution).

The Metropolis-Hastings algorithm: overview

The most popular MCMC method

- We are given $\vec{\rho}$, and looking for a suitable Q
- We split the transitions in two parts: "proposal" and "acceptance"
- At each step, we [stochastically] propose a move towards a different configuration
  - The proposal matrix can be chosen almost arbitrarily. Generally we propose moves from a configuration to a "nearby" one, and also keep it simple.
- Then we [stochastically] decide whether to accept the proposal. Refusing the move means not moving, which is the same as transitioning to the current state.
  - The acceptance rule cannot be completely arbitrary.
  - Metropolis-Hastings provides a simple and effective formula for the acceptance rule.

Proposal matrix has to be a stochastic matrix (similarly to Q) whose diagonal entries equal 0 (so A as well contrary to Q) (“transitioning to the current state" is when the move is refused, when it is accepted the state is changed according to the proposal). Otherwise it can be chosen almost arbitrarily (but it also makes sense to propose moves to nearby configurations).

- There are a few conditions that need to be enforced [required for *ergodicity*; see notes]:
  - No diagonal terms — **Always propose a change**
  - Connectedness — **It must be possible to get to any configuration from any other, given enough steps, with non-zero probability**
  - Aperiodicity — **There must be "enough" randomness not to get trapped in cycles**

Suggestions (local moves and others):

- Propose moves "close" to the current configuration
- Try to keep it simple (e.g. symmetric, uniform, ...)
- Try to get things to "mix up" well (extremely heuristic and vague, I know.. we'll see examples.)

Acceptance rule (Metropolis-Hastings rule):

$$A_{ij} = \min\left(1, \frac{C_{ji}\rho_i}{C_{ij}\rho_j}\right)$$

Guarantees detailed balance

Guarantees detailed balance (take Cjipi>Cijpj), then Qij=CijAij=Cij*1 and Qji=CjiAji=Cji*(Cijpj/Cjipi)=Cijpj/pi and we have Qji=Cijpj/pi=Qijpj/pi, so Qji=Qijpj/pi, so Qjipi=Qijpj. You can encounter situation of division by 0.

If the proposals are symmetric (Metropolis rule):

$$A_{ij} = \min\left(1, \frac{\rho_i}{\rho_j}\right)$$

Use either one or the other rule for all entries of A, for symmetric matrix you can use Metropolis rule and for non-symmetric matrix you have to use the general rule for non-symmetric proposals.

Properties of Q

Given the stationary condition (we consider Q such that p is its eigenvector, the situation where we reached the stationary distribution):

$$\vec{\rho} = Q\vec{\rho}$$

- Global balance condition (necessary and sufficient, not restrictive enough):

$$\forall i: \quad \sum_{j \neq i} Q_{ji}\rho_i = \sum_{j \neq i} Q_{ij}\rho_j$$

For each configuration: total "outflow" of probabilities equals the total "influx" of probabilities

Equivalent to stationarity condition. Easy to derive, since these sum to 1 in the presence of diagonal terms and if we subtract the same value from both sides, 1-Qiipi=1-Qjjpj because Qii=Qjj and pi=pj when i=j. DERIVATION? LOOK AT CS NOTES![2]

- Detailed balance condition (not necessary and sufficient):

$$\forall i, j: \quad Q_{ij}\rho_j = Q_{ji}\rho_i$$

The probability flows are now balanced for every pair of configurations

We can determine non-diagonal terms of Q given proposal C and acceptance A matrices, constructed according to specific rules. Then:

- Split the non-diagonal part in two: "proposal" C_{ij} and "acceptance" A_{ij}

$$Q_{ij} = C_{ij}A_{ij} \quad \text{if } i \neq j$$

The diagonal terms are set by the stochasticity property!

Transition probability from a given state to a given state equals the product of proposal of such move and acceptance of such move. And diagonal terms of Q can be computed by the stochasticity property (columns sum up to 1).

Having determined Q, you can further verify that p is an eigenvector of Q corresponding to eigenvalue 1, just for certainty.

Transition probabilities in a matrix Q: Qji =P (i → j) (rows are destinations and columns are sources).

$$\text{given } \vec{\rho}, \text{ find } Q \text{ such that: } Q\vec{\rho} = \vec{\rho} \qquad (2)$$

Let's rewrite this condition. Using the normalization condition (1) and isolating the diagonal terms in the matrix Q:

1. The first line is implies by the stationarity condition: Qp=p implies Qijpj=pi.
2. In the second line we rewrite the LHS of the above equation (Qijpj) explicitly, also isolating diagonal and non-diagonal terms, while for the RHS we rewrite as 1*pi=column sum (which is equal to 1 from the stochasticity property)*pi.
3. Third line we again rewrite the equation above with isolated diagonal terms.
4. We arrive at the global balance condition, which is necessary, implied by the stationarity condition (we went from the stationarity condition).

[2] Rewrite using indices explicitly, isolate the diagonal term, use the stochasticity property, rearrange (see notes)

$$\forall i: \quad \sum_{j=1}^{M} Q_{ij}\rho_j = \rho_i$$

$$\sum_{j\neq i} Q_{ij}\rho_j + Q_{ii}\rho_i = \left(\sum_{k=1}^{M} Q_{ki}\right)\rho_i$$

$$\sum_{j\neq i} Q_{ij}\rho_j + Q_{ii}\rho_i = \sum_{k\neq i} Q_{ki}\rho_i + Q_{ii}\rho_i$$

We arrive at this expression:

$$\forall i: \quad \sum_{k\neq i} Q_{ki}\rho_i = \sum_{j\neq i} Q_{ij}\rho_j \tag{3}$$

Global balance is equivalent to stationarity condition. Detailed balance is more restrictive, if detailed balance holds, global holds as well. The interpretation is that the probability flows are now balanced for every pair of configurations.

Split the off-diagonal transition probabilities Qij into a proposal part Cij and an acceptance part Aij. Proposal matrix should be normalized along columns and zero on the diagonal. Starting from some state i, we always propose some another state j. Diagonal terms of matrix A are irrelevant.

The diagonal case for the matrix Q: the probability of not accepting the proposed move.

Symmetric proposals, exponential reparametrization, acceptance rules

$$C_{ij} = C_{ji}$$

We also define the quantities u_i, Δ_{ij} by:

$$\rho_i = \frac{1}{Z}e^{u_i}$$

$$\Delta_{ij} = u_i - u_j$$

$\Delta_{ij} = -\Delta_{ji}$.

$$\ln A_{ij} \quad = \quad \frac{1}{2}\left(\Delta_{ij} + g\left(\Delta_{ij}\right)\right) \tag{9}$$

Since A_{ij} must be a probability, and thus less than 1, the logarithm must be negative and we obtain this condition on g:

$$\forall x: \quad g\left(x\right) \leq -x \tag{10}$$

With these definitions, the detailed balance condition eq. (4) becomes:

$$\forall i,j: \quad e^{\frac{1}{2}\left(\Delta_{ij}+g\left(\Delta_{ij}\right)\right)}e^{u_j} = e^{\frac{1}{2}\left(-\Delta_{ij}+g\left(-\Delta_{ij}\right)\right)}e^{u_i}$$

which after simple algebraic manipulations simplifies to the requirement:

$$\forall x: \quad g\left(x\right) = g\left(-x\right) \tag{11}$$

This also implies that the bound of eq. (10) becomes stricter:

$$\forall x: \quad g\left(x\right) \leq -\left|x\right| \tag{12}$$

---

[2] Mathematically, this doesn't work for those i where $\rho_i = 0$, but this is just a minor annoyance and not a substantial problem: just expunge those configurations from X and start over. Thus assume that $0 < \rho_i \leq 1$.

[3] Here's one way to find a valid vector $\vec{u}$: set $u_1 = 0$, then for all $i \geq 2$ set $u_i = u_1 + \ln\left(\rho_i/\rho_1\right)$. It may seem surprising, but you can check that it works by substituting it back into eq. (7).
Suppose now that you have found a valid $\vec{u}$ and consider an alternative $\vec{u}'$ with $u'_i = u_i + c$, i.e. shift the whole vector $\vec{u}$ by a constant quantity. This produces the same probability vector $\vec{\rho}$, since $\frac{e^{u'_i}}{\sum_k e^{u'_k}} = \frac{e^{u_i+c}}{\sum_k e^{u_k+c}} = \frac{e^c e^{u_i}}{e^c \sum_k e^{u_k}} = \frac{e^{u_i}}{\sum_k e^{u_k}} = \rho_i$. Furthermore, $\Delta_{ij} = u_i - u_j = u'_i - u'_j$ is also unaffected by the shift. Therefore we say that eq. (7) defines $\vec{u}$ uniquely up to an additive constant.

We have finally arrived at the following result: as long as we can find any function $g(x)$ that is symmetric (criterion (11)) and bounded as in (12), we can then use it to build a matrix A_{ij} using eq. (9). Then we can pick (almost arbitrarily[4]) a symmetric choice matrix C and finally obtain a matrix Q that satisfies detailed balance, eq. (4), and thus our stationarity condition, eq. (3), and therefore solve our original problem (2).

We still have to choose g. Here are two common choices (the first one being by far the most common):

1. Metropolis rule: $g(x) = -|x|$. This is obviously the simplest one. Then

$$A_{ij} = e^{\frac{1}{2}(\Delta_{ij} - |\Delta_{ij}|)} = \begin{cases} 1 & \text{if } \Delta_{ij} \geq 0 \\ e^{\Delta_{ij}} & \text{if } \Delta_{ij} < 0 \end{cases}$$
$$= \min\left(1, e^{\Delta_{ij}}\right) = \min\left(1, \frac{\rho_i}{\rho_j}\right) \tag{13}$$

2. Glauber (AKA Berker) rule: $g(x) = -2\ln\left(2\cosh\frac{x}{2}\right)$. Then

$$\begin{aligned} A_{ij} &= e^{\frac{1}{2}\left(\Delta_{ij} - 2\ln\left(2\cosh\frac{\Delta_{ij}}{2}\right)\right)} \\ &= \frac{e^{\frac{1}{2}(u_i - u_j)}}{e^{\frac{1}{2}(u_i - u_j)} + e^{-\frac{1}{2}(u_i - u_j)}} \\ &= \frac{e^{u_i}}{e^{u_i} + e^{u_j}} = \frac{\rho_i}{\rho_i + \rho_j} \end{aligned} \tag{14}$$

Metropolis rule is generally preferable, since it will have the highest possible acceptance rate compatible with the bounds and the detailed balance condition (in the symmetric proposals scenario).

above, we generally impose some additional mild conditions on C. This is because we normally want to ensure that the resulting matrix Q has $\vec{\rho}$ as its *only* stationary state, because we want to ensure that, for *any* initial vector of probabilities $\vec{v}$, the Markov chain process converges to our desired stationary limit, $\lim_{n\to\infty} Q^n \vec{v} = \rho$. Mathematically, this can be ensured by requiring that C has only one eigenvector with eigenvalue 1. One important condition that

SIMULATED ANNEALING INTRO

With our notation above, we see that we can simply set $u_i = -\beta E(i)$ and then derive the Metropolis rule as $A_{ij} = \min\left(1, e^{-\beta(E(i) - E(j))}\right)$. Recall that in this notation j is the starting configuration and i the proposed move, thus $\Delta E_{ij} = E(i) - E(j)$ is the comparative "cost" of accepting the move. If it is negative than the move is convenient and it is always accepted, if it is positive than it is only accepted with some probability that depends on β and rapidly becomes lower as β increases, going to 0 as $\beta \to \infty$.

As a matter of computational efficiency, it is also important to point out that at each step of MCMC we will have some current configuration j and a proposed (different) configuration i extracted according to C_{ij}, and we will need to compute $\Delta E_{ij} = E(i) - E(j)$. We would like this computation to be cheap so that we can perform as many MCMC steps as possible. In many cases, this occurs when the configurations i and j are sufficiently "similar" (what this means de-

given j, set $C_{ij} = 0$ for all i which are not "similar" to j in this sense. Typically this means that only a tiny fraction of the entries of C are going to be non-zero. As a result our MCMC always moves at most by "small increments" at each step.

Very often, it is also the case that this choice of only proposing "local" moves in which the configuration is not changed too much is also crucial to have reasonably good chances that the proposed move will be accepted: if we

The next observation is about what happens for different values of β. At $\beta = 0$, the distribution ρ_i is just uniform. If we run MCMC we are just going to accept all moves, and thus perform a so-called "random walk" in the set X:

$E(i)$ at all. At the other extreme, when β is very large and tends to infinity, the probability ρ_i becomes very sharply concentrated on the minima of the cost $E(i)$. To see this, call $i^\star$ the configuration with the minimum cost, $i^\star = \arg\min_i E(i)$, and suppose that it is unique. Then write

$$\lim_{\beta\to\infty} \frac{e^{-\beta E(i)}}{\sum_k e^{-\beta E(k)}} = \lim_{\beta\to\infty} \frac{e^{-\beta E(i)}}{e^{-\beta E(i^\star)} + \sum_{k\neq i^\star} e^{-\beta E(k)}}$$

$$= \lim_{\beta\to\infty} \frac{e^{-\beta(E(i)-E(i^\star))}}{1 + \sum_{k\neq i^\star} e^{-\beta(E(k)-E(i^\star))}}$$

$$= \begin{cases} 1 & \text{if } i = i^\star \\ 0 & \text{otherwise} \end{cases}$$

The more non-local the moves, the larger the space of candidate moves, the higher the chance that our proposed move gets us to a random configuration that makes no sense.

Our MCMC process would still be more likely to accept moves that decrease the current cost as opposed to those that increase it (this is more and more true for large beta). However, it would also have a chance to escape from local minima (this is less and less true for large beta though). The idea of Simulated Annealing is to perform a MCMC on the Gibbs distribution starting at small and gradually increase it. (The process of increasing beta is called annealing.) Building up a good configuration incrementally, by starting at random and fixing the configuration a piece at a time.

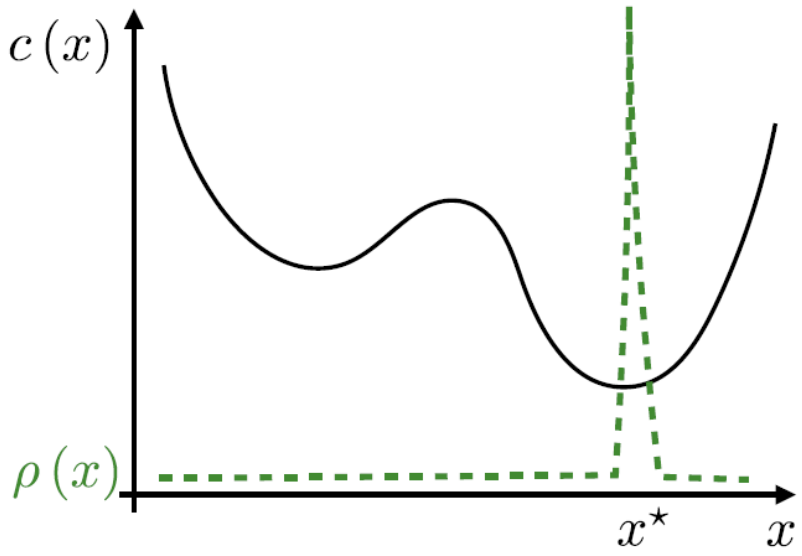
- Given a cost function E, choose a suitable representation for the configurations of the problem
- Choose the kind of moves that we want to propose (i.e. choose how to set the elements C_{ij} keeping in mind that we always need to set the diagonal elements to 0). Possibly, in such a way that computing the cost differences ΔE_{ij} is cheap, ideally $O(1)$.
- Choose how to set the initial configuration (normally, it's just at random)
- Choose an "annealing protocol": A sequence of increasing values of β, and a number of MCMC steps to perform for each value of β.

Simulated Annealing

It is another instance of MCMC, a method based on stochastic processes and MCMC. We will be simulating the annealing process with MCMC. Before we were sampling from a given probability distribution using MCMC. Now we want to find the distribution centered around the minimum, we

do so by adjusting proposals and acceptance accordingly to increase the likelihood of ending up in the minimum.

- Given a distribution $\rho(x)$, we can use MCMC to sample from it
- But we are given a cost function $c(x)$ to minimize instead
- Can we find a $\rho(x)$ which is concentrated in the minimum of $c(x)$?



We want the probability of x to peak at the minimizer of c, so where c(x) is minimum.

Lowering the temperature to reach the atomic configuration x corresponding to the state of minimum energy E(x) (minimizing the internal energy). It must be done gradually (annealing is about letting it cool slowly). Increasing the beta parameter (which is the inverse of temperature) gradually (or increasing?) to minimize the cost function (internal energy).

Cons: – it's heuristic, no (useful) guarantees; – requires some judgment, trial and error, fiddling with parameters; Pros: – very general, not too hard to implement; – often works quite well; – interesting programming techniques.

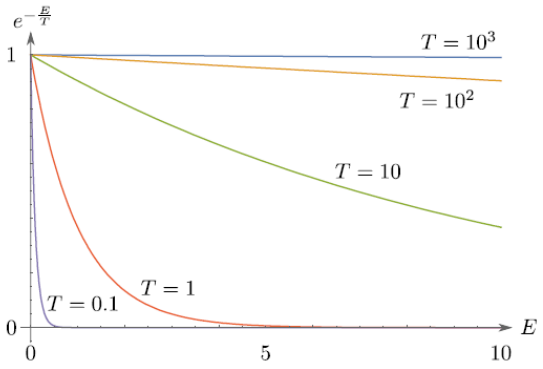
$P_T(x)$ = "Prob. to find the atoms arranged as x (at temperature T))". The lower the temperature, the more likely that an x picked at random will have low energy.

- **Boltzmann distribution**: $P_T(x) = \frac{1}{Z_T} e^{-\frac{1}{T}E(x)}$
- Z_T: normalization constant, to ensure $\sum_{x \in \mathcal{X}} P_T(x) = 1$

Properties of Boltzmann distribution:

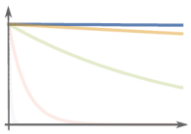
$$P_T(x) = e^{-\frac{1}{T}E(x)} / \sum_{x' \in \mathcal{X}} e^{-\frac{1}{T}E(x')}$$

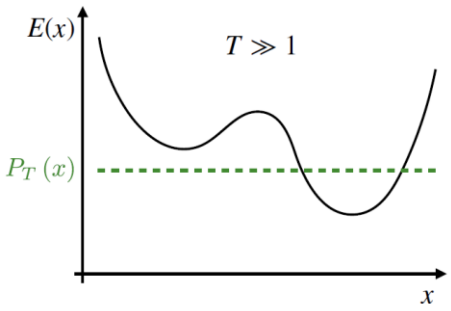
- Denominator
  - Constant
  - Could be hard to compute (many terms)
- Numerator
  - Monotonically decreasing with the energy
  - Almost constant for large T
  - Effect is more pronounced for small T



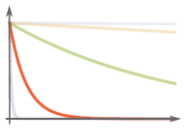
For very high temperatures, the probability is basically uniform for different energy states (remember we divide the quantity on the y-axis of the graph above by normalizing constant!)

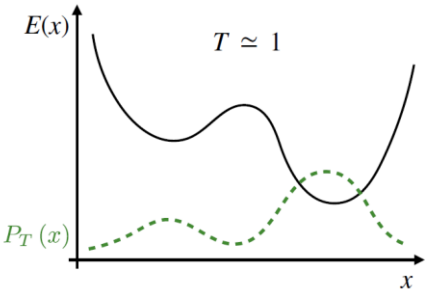
- Very high temperature: basically uniform



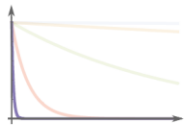


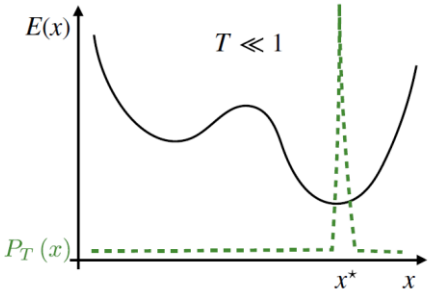
- Intermediate temperature: lower energy has higher probability





- Very low temperature: probability concentrated on the global minimum





- At very high temperature, $T \to \infty$, the argument of the exponential goes to 0 and all the x have the same probability: $P_\infty(x)$ is uniform
- At very low temperature, $T \to 0$, the probability concentrates on the global minimum of $E(x)$. Say that $x^\star$ is the optimum (say it's unique), and consider

$$P_T(x) = e^{-\frac{1}{T}(E(x)-E(x^\star))} / \sum_{x' \in \mathcal{X}} e^{-\frac{1}{T}(E(x')-E(x^\star))}$$

Only the terms with $x \approx x^\star$ survive at low T

- In between, we have intermediate situations: a higher probability on low-energy states, a lower probability on high-energy states. But the low-energy states are often few so the probability of getting some medium energy state is larger just because there are more of those, even if individually they are less likely.

Optimization scheme

Use c(x) as an energy in the Boltzmann formula. For any T, obtain $P_T(x)$.

Sampling from $P_T(x)$, Boltzmann distribution (which has a useful property of higher probability of lower-energy for intermediate and lower temperatures**) by doing MCMC and using Metropolis-Hastings rule** (directly is hard because ZT is hard to compute, it is a sum of an exponential number of terms). We know that the lower the temperature $P_T(x)$ is higher for lower-energy configurations (that is why we want to randomly sample from it as it gives us a higher chance of arriving at the lower-energy state than a uniform distribution). For higher temperature (lower beta), we have a relatively high acceptance rate (probability of states is uniform) and we explore the space, while for lower temperature (higher beta), the algorithm is more selective and concentrates more on a smaller area which it detected as potentially containing the minimum throughout previous steps. We want to cool gradually to be able to arrive at the area with global minimum instead of instantly jumping risking getting trapped in a local minimum. Sometimes we accept moves that increase the cost. We want to combine broad space exploration with the precision of greedy.

WE WANT TO STRIKE THE RIGHT BALANCE BETWEEN BREADTH OF THE EXPLORATION AND "DEPTH", SPEED OF CHANGING TEMPERATURE (WE DO NOT WANT TO CHANGE IT TOO FAST BECAUSE WE MAY GET STUCK IN LOCAL MINIMUM NOR TOO SLOW BECAUSE IT IS COSTLY TO RUN THE ALGORITHM FOR UNNECASSARILY LONG TIME). PRECISION OF ESTIMATE VS ALGORITHM'S RUNNING TIME – TRADE-OFF.

Proposal scheme: local moves, requirements: x' cannot be the same as x; connectedness (must be able to cover all X when jumping around); aperiodicity (must be random enough). For simplicity, we may assume symmetry[3] so we can use Metropolis simplified.

Acceptance: depends on delta cost and beta.

$$A(x \to x') = \min\left(1, \frac{\rho_{x'}}{\rho_x}\right) = \min\left(1, \frac{\frac{e^{-\frac{1}{T}c(x')}}{Z_T}}{\frac{e^{-\frac{1}{T}c(x)}}{Z_T}}\right) = \min\left(1, e^{-\frac{1}{T}\left(c(x')-c(x)\right)}\right)$$

$$A(x \to x') = \min\left(1, e^{-\beta \Delta c(x \to x')}\right)$$

- When $\Delta c \leq 0$ (the move reduces the cost) the min is 1, acceptance is certain
- When $\Delta c > 0$ (the move increases the cost) the min is < 1, acceptance is likely only if move not-too-bad (depends on β, small→everything goes, large→picky)

When delta_cost>=0, there is still a positive probability of accepting a move, if delta_cost is small then this probability is higher than for big delta_cost. Also it depends on beta.

"Annealing schedule"[4]: a sequence of increasing betas (decreasing temperatures) and corresponding number of MCMC iterations.

Process:

1. Start from a random initial configuration. Start at small beta (large T) and sample (run MCMC long enough): propose and accept moves from a given configuration.
2. Increase beta (lower T) and keep going, starting from where you had left and running for more iterations.
3. Do the same for all betas every time starting from where you had left (**better initialization which helps to avoid getting stuck in local minimum**).
4. By the point you get to beta→inf (T→0), you're now running the greedy algorithm. But now you are initializing it from a configuration sampled from a previous $P_T(x)$, not so randomly.
5. Ideally, in the limit of infinite MCMC samples and infinitely slow annealing you'd be guaranteed to find the optimum but it is not feasible.
6. Instead of returning the last configuration, keep the best one seen throughout the process.

[3] Chance that x' gets proposed given x must be the same as the chance that x gets proposed given x'.
[4] Simplified: The first L-1 steps are equally spaced between β0 and β1. The last one is performed at β=np.inf. All the annealing steps will have the same number of MCMC iterations, for example around 100 per annealing step.

We'll need to monitor and output the "acceptance rate": at every annealing step, what fraction of the proposed moves have been accepted? This will be often crucial for setting the parameters. Tuning by trial-and-error to go just as slowly as necessary. Rule of thumb: set beta_0 (the first, smallest beta) such that the acceptance rate is between 30% and 80% and we want it to decrease gradually to reach a few percent for penultimate step. Too low early (risk of local minimum), too high at the end (wasted time, instead of optimizing). Sometimes it is worth it to increase the number of MCMC iterations.

Pseudocode:

- **INPUTS:** optimization problem over X; annealing schedule $[(\beta_i, it_i)$ for i in $\{0,..,L-1\}]$
- **ALGORITHM:**

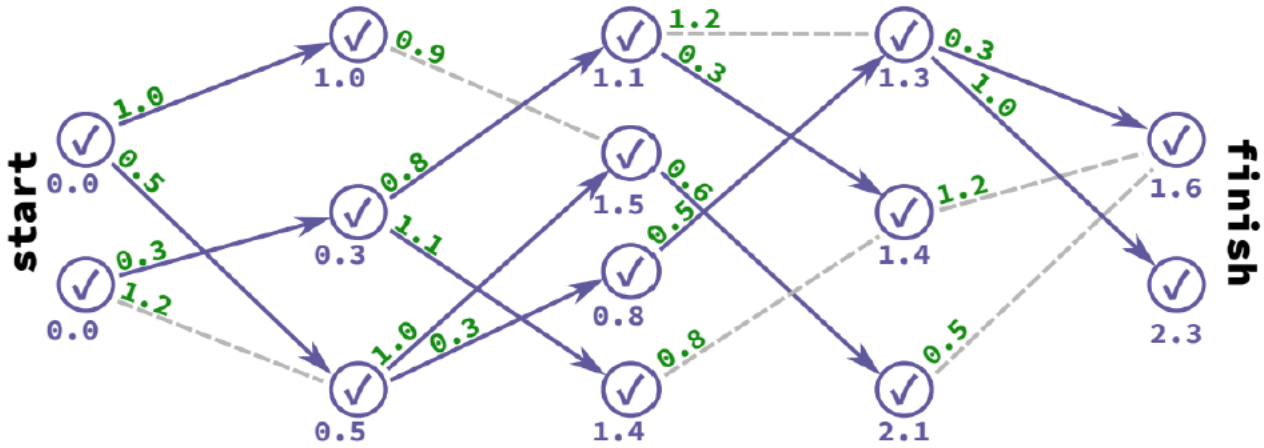
```
initialize x in X at random
for i in 0,1,…,L-1:    # annealing step
    set β = β_i        # current β (inverse temperature)
    for t in 1,…,it_i:  # perform this many MCMC steps
        given x, produce a proposed move x'
        compute the cost difference Δc(x→x') # c(x')-c(x)
        compute the acceptance rate A(x→x') from Δc(x→x') and β
        with probability A(x→x'):
            set x = x'  # accept the proposal
return x
```

Dynamic Programming

Used to solve (discrete) combinatorial optimization problems in polynomial time.

Optimal left-to-right path in a layered directed acyclic graph. A configuration is the result of a sequence of decisions and partial configurations (incomplete decisions) have a well-defined cost. The optimal decision can be built from optimal partial decisions.

Given a node, what's the min cost to get there? We ask this question for the last layer and then recur backwards.



Forward pass to find the optimal cost and backtracking using whence structure.

Recursive problem structure:

– Given a node, the best cost to reach it is the minimum among the best costs at the previous layer plus the cost of the final step to the node [recursive step]

– Exception: nodes at the start layer: the cost to reach those is just zero [base case]

The recursive structure (an optimal path is made up from optimal partial paths) allows the re-use of previous computations, which makes the problem polynomial (the amount of optimal partial configurations is polynomial – one per node). Every time you apply the recursive relation you take a minimum over a polynomial number of terms, a number of computations proportional to the number of links. We update the costs all the time.

Top-down approach: recursion and memorization.

Bottom-up approach: pre-allocate the space for the data you need.

- Recursive (Top-down) approach
  - Memoize the partial costs and the partial best solutions
  - Compose the partial solutions in the recursive step
- Bottom-up approach:
  - Forward pass: compute partial costs + the decision made to achieve it
  - Backward pass: backtrack the best configuration

The bottom-up approach is generally preferred because it avoids the overhead of function calls

Seam carving: we seek the lowest-content (less overall edges, optimal) seam. An edge-detection filter is applied to the image to determine where the content is and it is the input (array with these costs) to our dynamic programming algorithm.

base case: $c_{0j} = g_{0j}$

recursive relation: $c_{ij} = g_{ij} + \min\{c_{(i-1)(j-1)}, c_{(i-1)j}, c_{(i-1)(j+1)}\}$ if $i > 0$

g

	0	1	2	3
0	1.0	1.0	0.5	0.0
1	2.0	1.0	1.5	3.0
2	0.0	0.5	1.5	2.0
3	0.0	0.75	1.0	0.5
4	1.0	0.5	0.5	1.0

row 0 copied

c

	0	1	2	3
0	1.0	1.0	0.5	0.0
1	3.0	1.5	1.5	3.0
2	1.5	2.0	3.0	3.5
3	1.5	2.25	3.0	3.5
4	2.5	2.0	2.75	4.0

1.0 + min(2.0, 3.0, 3.5)

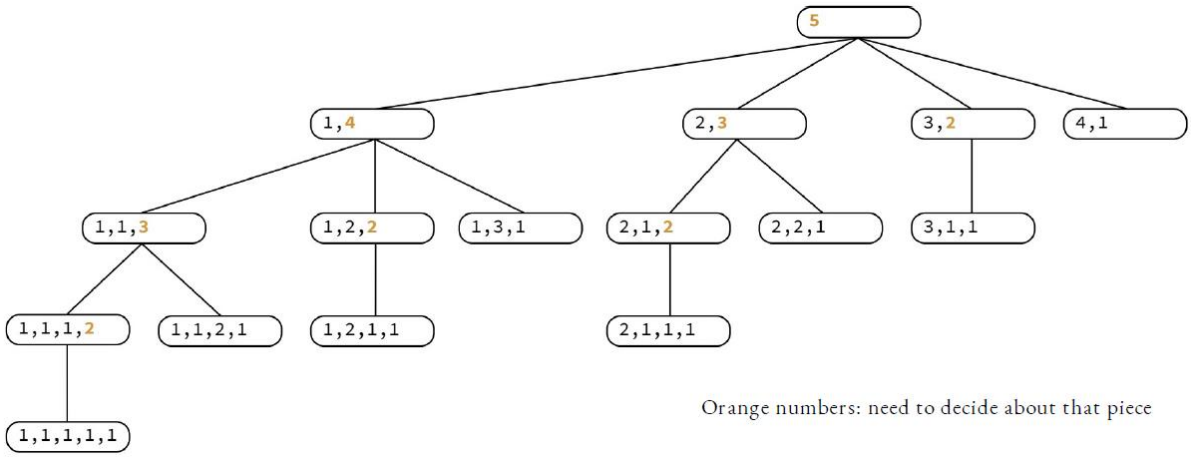
C is matrix of cumulative costs.

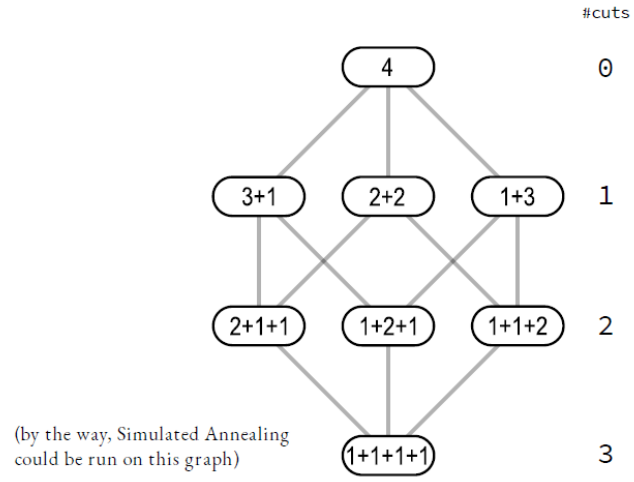
whence

	0	1	2	3
0				
1				
2				
3				
4				

Rod-cutting: given n and the prices pi, you want to find the way to cut the rod that maximizes your revenue. Configuration space: number of possible cuts is 2^{n-1} (compositions), number of distinct cuts is also exponential (partitions).

Top-down approach: recursion combined with memoization.





At each moment, you've got to decide whether to – not cut, or – cut a piece of length 1, or of length 2, or 3, etc. then work on the remainder. Maximum over sum of the revenue from the part which we keep fixed and the maximum revenue which we can get from the rest (recursion). Revenue of cutting a piece of length i with a remainder of n - i = pi + whatever you can get from the remainder.

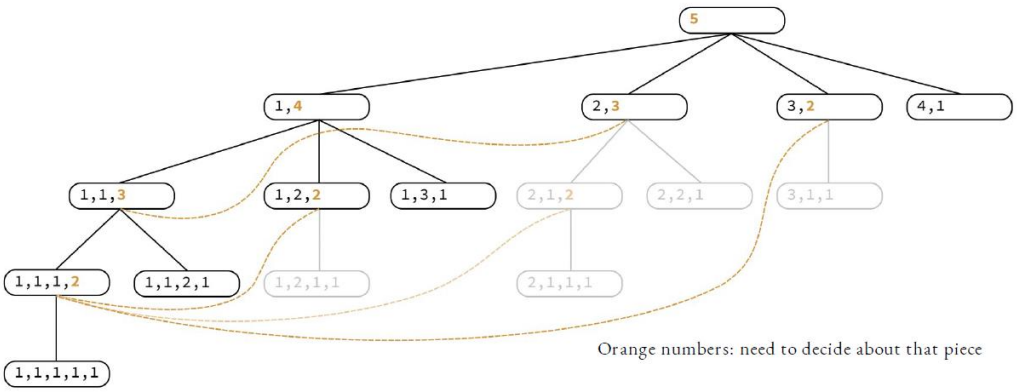
- The best overall is thus $r_n = \max\left(p_n, \max_{1 \le i < n} (p_i + r_{n-i})\right)$

base case: $r_0 = 0$

recursive relation: $r_j = \max_{1 \le i \le j} (p_i + r_{j-i}) \quad \text{if } j > 0$

Size of decision tree is exponential and recursive approach goes through all of it. However, we can do better since some choices are essentially the same (just different order, different composition but the same partition). If we store the results, we could skip the entire subtrees. The pruned tree is $O(n^2)$. We need to combine it with memoization, complexity O(n) as we just store results for $r_1,\ldots r_n$.

We also want argmin so we can have a whence structure as an array of int which says the length of the piece which was kept fixed for a given rj. Then we can backtrack.



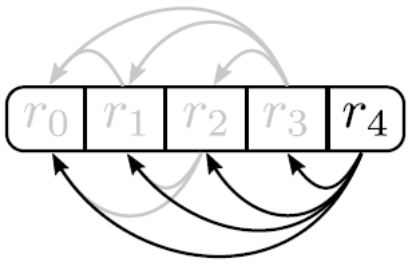
Bottom-up:

- The recursive formula is still the same, but we read it "right-to-left"

$$\text{base case:} \quad r_0 = 0$$

$$\text{recursive relation:} \quad r_j = \max_{1 \le i \le j} (p_i + r_{j-i}) \quad \text{if } j > 0$$

Forward pass: Keep an array of length n+1 with max , fill the base case for j=0 , then compute from j=1 up. For each j, there is going to be an i* corresponding to the argmax: whence.



Backward pass: reconstruct backwards, starting from j=n. Whence[j] tells you: the size of the cut, call it I and that your next cut can be found in whence[j-i].

Divide and conquer: decide the first cut, obtain 2 sub-problems (left and right) to be solved independently. Iterate. Do that for all possible divisions, you can't just pick one. Joining two optimal solutions of size i and j gives you an optimal solution of size i+j under the condition that there is a cut at i (or j).

Generalize: if you knew how to optimally cut a length i, for all i from 1 up to some j-1 , you could just attempt all combinations i+(j-i) and see which is best, and if it is better than leaving the length j uncut. Joining the known solutions to sub-problems with extra portion from the bigger instance of the problem in all possible ways.

Floyd-Warshall algorithm

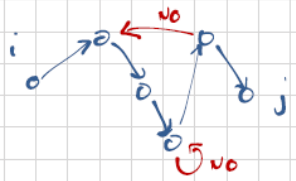
Algorithm to find all-pairs shortest paths for a directed, weighted graph with n nodes, with possibility of missing links, with possibility of negative costs but without negative cycles.

- Call w_{ij} the cost of the link $i \to j$ (may be ∞)
- Call c_{ij} the optimal cost of the path $i \to \dots \to j$ (may be ∞)

unknown!

(1) $\forall_{i,j}: c_{ij} \leq w_{ij}$

(by definition)



(2) Along the optimal path, a node can be touched at most once

- otherwise there would be a cycle, which has cost ≥ 0

$\to$ implies longest opt. path has length n

2 derives from non-negative cycles.

- If the optimal path from i to j passes through some intermediate node k $\quad i \to \dots \to k \to \dots j$
- then the pieces $\begin{cases} i \to \dots \to k \\ k \to \dots \to j \end{cases}$ must be optimal for connecting $\begin{cases} i \text{ to } k \\ k \text{ to } j \end{cases}$

prove by contradiction!

- Thus the cost of the optimal path $i \to j$ must be

either: w_{ij}

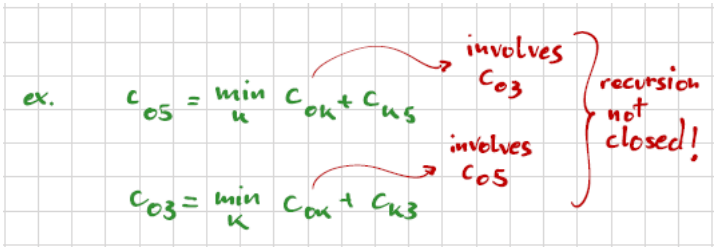
or: $\min_{k \neq i,j} c_{ik} + c_{kj}$

but since $c_{ii} = 0$, $c_{jj} = 0$ $\Rightarrow$ $c_{ij} = \min_{k} (c_{ik} + c_{kj})$

Recursive relation:

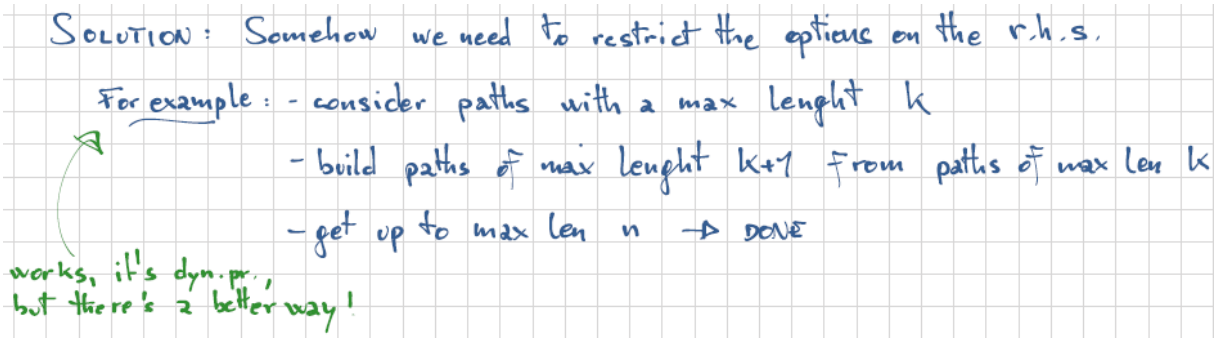
$$c_{ij} = \min_{k=0 \dots n-1} c_{ik} + c_{kj}$$

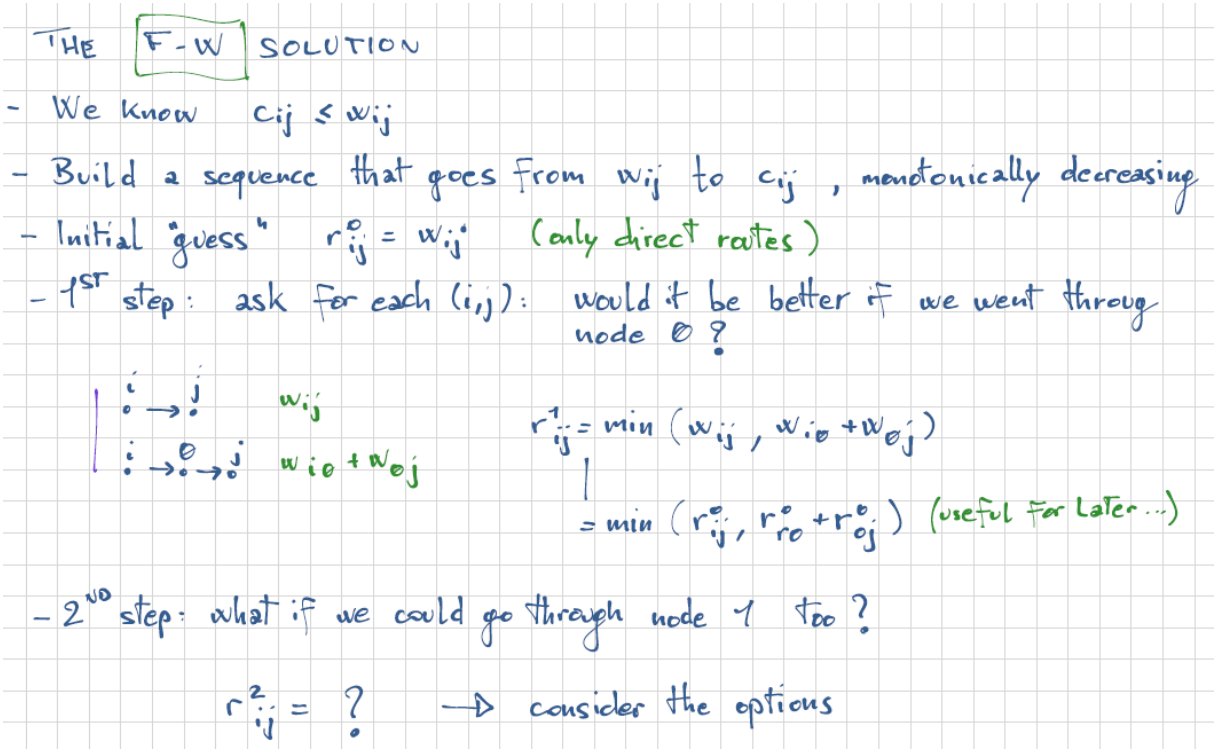
Recursion is not closed, no base case:

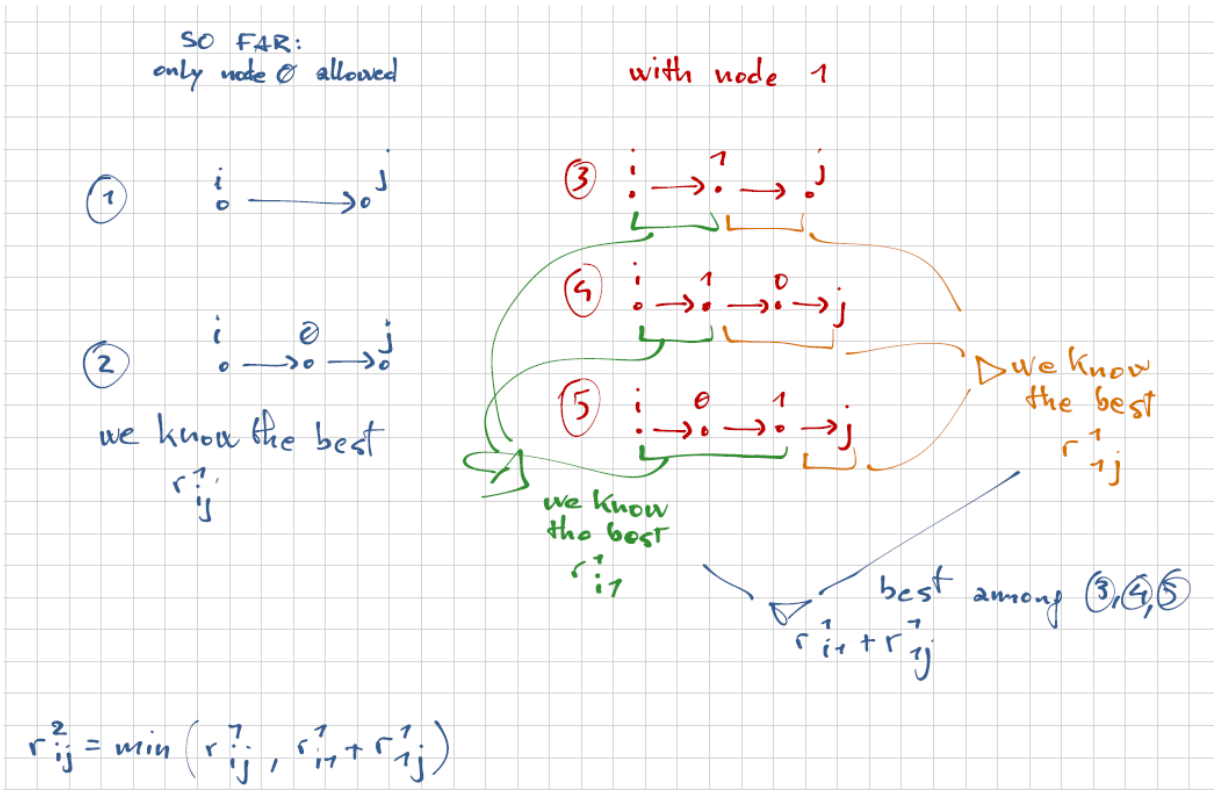


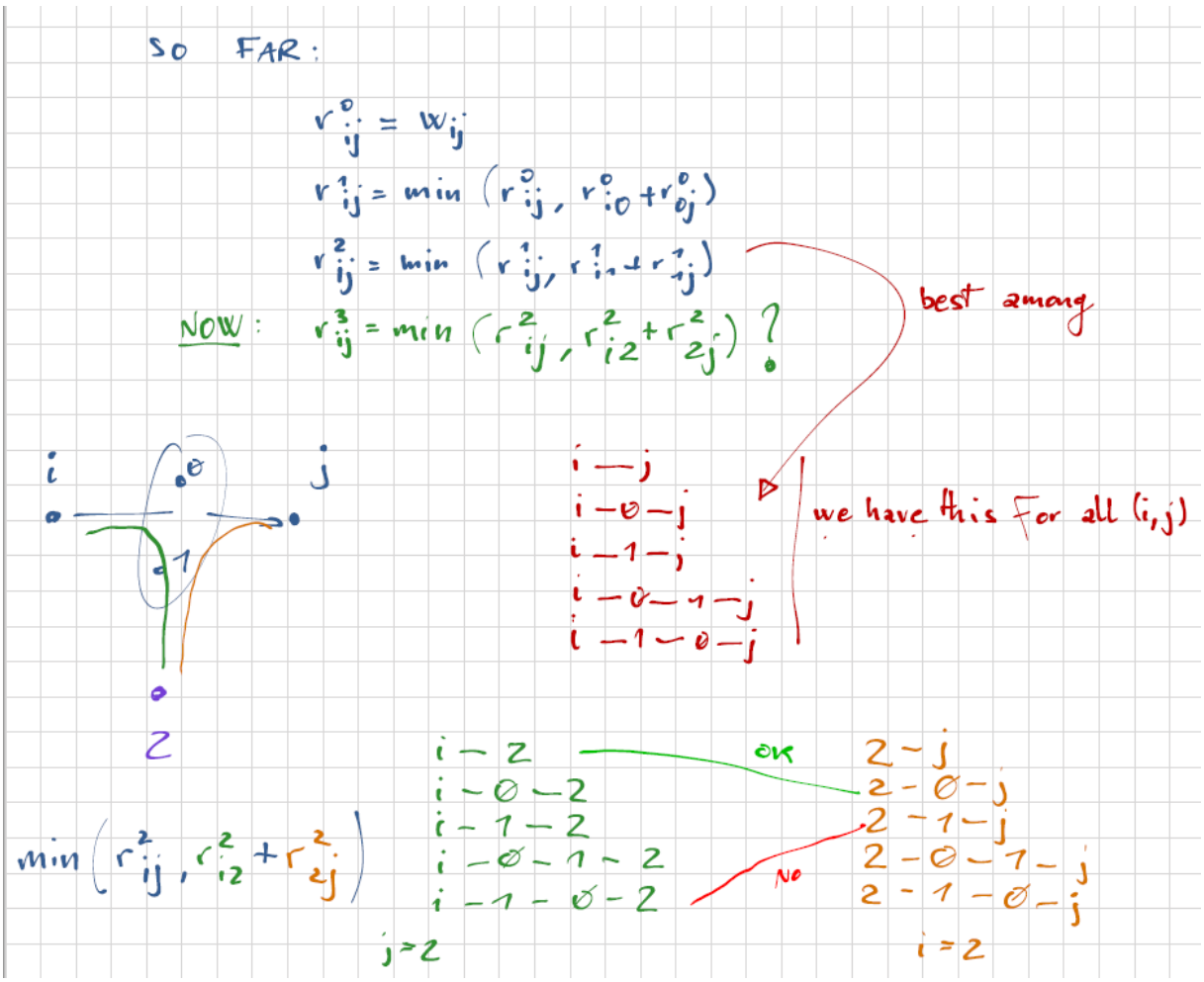
But Floyd-Warshall algorithm helps as it allows to create some order to our computations.

Alternative idea (less efficient than Floyd-Warshall, n^3?) to close the recursion:






SO FAR:
only node ∅ allowed
with node 1
1
2
3
4
5
we know the best
r^1_{ij}
we know the best
r^1_{i1}
we know the best
r^1_{1j}
best among 3,4,5
$r^1_{i1} + r^1_{1j}$
$r^2_{ij} = \min\left(r^1_{ij}, r^1_{i1} + r^1_{1j}\right)$


SO FAR:
$r^0_{ij} = w_{ij}$
$r^1_{ij} = \min\left(r^0_{ij}, r^0_{i0} + r^0_{0j}\right)$
$r^2_{ij} = \min\left(r^1_{ij}, r^1_{i1} + r^1_{1j}\right)$
NOW: $r^3_{ij} = \min\left(r^2_{ij}, r^2_{i2} + r^2_{2j}\right)$?
best among
i — j
i — ∅ — j
i — 1 — j
i — ∅ — 1 — j
i — 1 — ∅ — j
we have this for all (i,j)
i
∅
j
1
2
$\min\left(r^2_{ij}, r^2_{i2} + r^2_{2j}\right)$
i — 2
i — ∅ — 2
i — 1 — 2
i — ∅ — 1 — 2
i — 1 — ∅ — 2
OK
NO
2 — j
2 — ∅ — j
2 — 1 — j
2 — ∅ — 1 — j
2 — 1 — ∅ — j
j = 2
i = 2

IN GENERAL $r^0_{ij} = w_{ij}$ base case

$$r^{k+1}_{ij} = \min\left(r^k_{ij},\ r^k_{ik} + r^k_{kj}\right)$$

$i \to j$ possibly through $0, 1, \ldots, k$

$i \to j$ $\quad$ $i \to k$ $\quad$ $k \to j$

possibly through $0, 1, \ldots, k-1$

$\forall i \quad w_{ii} = \emptyset$

$\Downarrow$

$\forall k \forall i \quad r^k_{ii} = \emptyset$

NOTE $r^{k+1}_{ij} \leq r^k_{ij} \quad \forall i,j,k$

WHEN DOES IT STOP?

when $k+1 = n$

$k = n-1$

at that point $r^n_{ij} = c_{ij}$

our goal!

BOTTOM-UP APPROACH

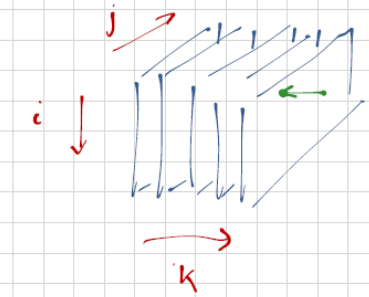
1. INITIALIZE A MATRIX $r = w$
2. ITERATE for $k = 0 \ldots n-1$ $\quad r[i,j] = \min\left(r[i,j],\ r[i,k] + r[k,j]\right)$

(we can overwrite the previous matrix)

③ KEEP TRACK OF THE ARGMIN! HOW?

we don't do this! →

- we could keep a whence structure similar to r^k_{ij}, (n x n x n array)
  with entries 0, 1, R
  - choose r^k_{ij}
  - choose $r^k_{ik} + r^k_{kj}$



- for a given ij
- start from whence [i,j,-1]
- reduce k until you hit a 1
- you found a node k, store it
- if k==i you're done
- otherwise recursively do the same for ik and kj

we do this instead →

- keep a matrix (n x n) of int, called pred

pred[i,j] = "what is the predecessor of j in the opt. path i→...→j"

WHY IS pred SUFFICIENT?

- keep a matrix (n x n) of int, called pred:

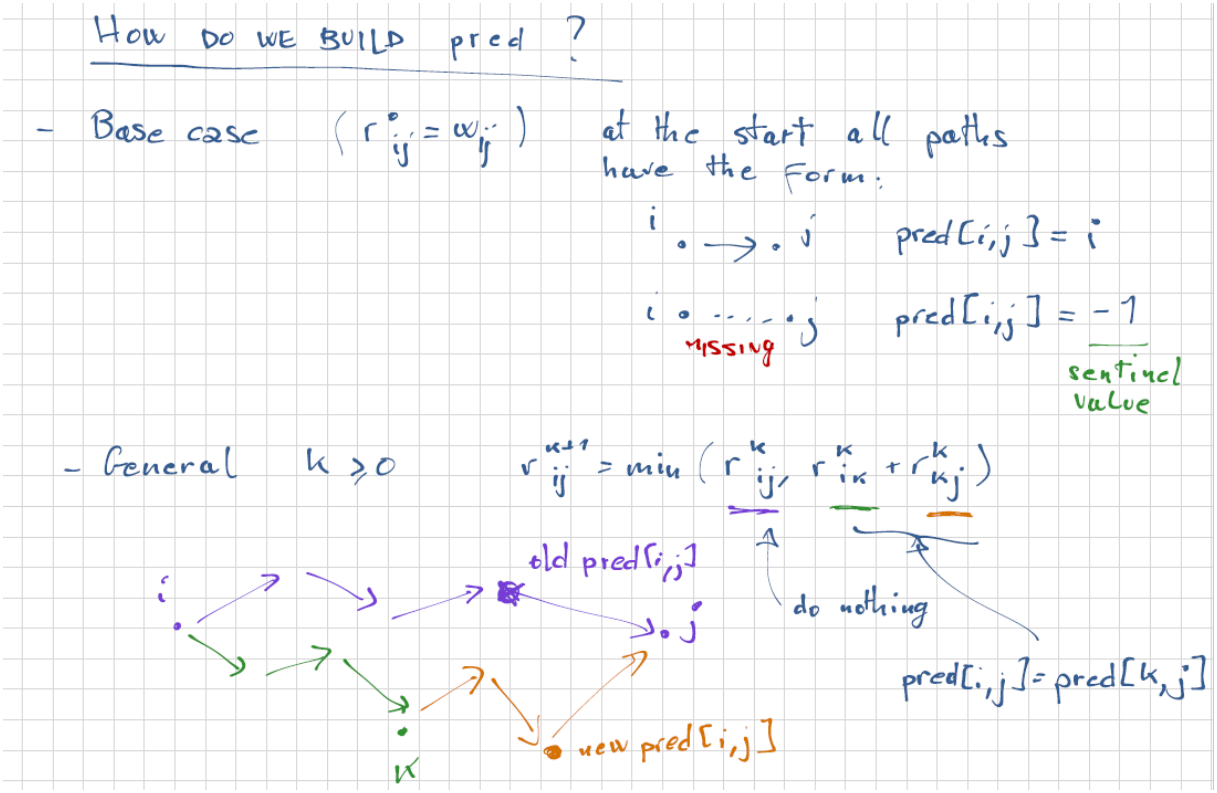
pred[i,j] = "what is the predecessor of j in the opt. path i→...→j"

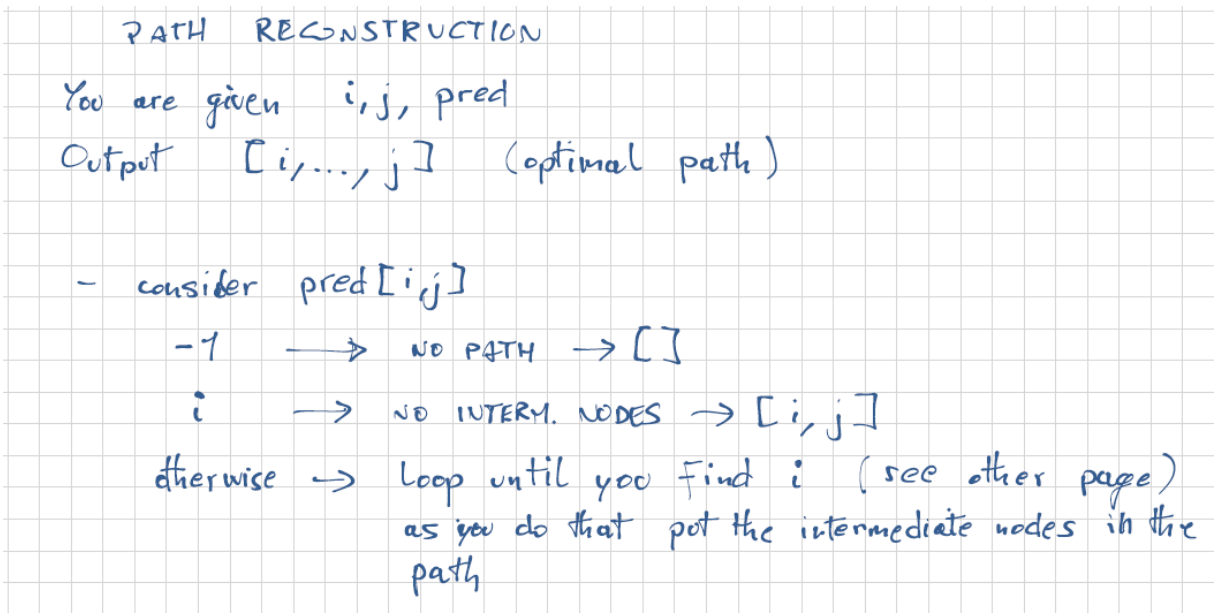
[i, j]
[i, z, j]
[i, z', z, j]
[i, z'', z', z, j]

as we walk the path back, we insert the nodes we found in the path at position 1

z = pred[i, j]
z' = pred[i, z]
z'' = pred[i, z']
i = pred[i, z'']

DONE!





Optimization

Gradient descent

1 dimension

Objective function, assumption: continuity, well-posed problem (minimizer is not at +/- infinity).

If f(x) is continuously differentiable (the derivative f0(x) exists and it is continuous in R) the global minimum x* is a stationary point, satisfying the condition: f'(x) = 0 (also for local minima, maxima,

saddles). A local minimum is a point which is a (global) minimum only if the space is restricted to a neighborhood of the point itself.

$$|f(x^{appr}) - f(x^{loc})| = f(x^{appr}) - f(x^{loc}) < \epsilon.$$

We would like to implement an *iterative procedure*, that starts from some initial conditions, $x^{(0)}$, and that updates at iteration k the proposal to $x^{(k)}$ through some rule that uses the information collected during the previous updates. k has the role of a time index in our iterative procedure.

Both $f(x^k)$ and $f'(x^k)$ are just *pointwise* pieces of information, but thanks to the regularity of $f(x)$ (e.g. continuity), they also tell us something on how the function $f(x)$ behaves in proximity of x^k. In particular, by the very definition of differentiability, we know that

$$f(x) = f(x^k) + f'(x^k)(x - x^k) + o(x - x^k). \tag{2.6}$$

This means that the remainder of the first order Taylor expansion goes to zero faster than $x - x^k$ when $x \to x^k$.

The *Taylor expansion* of order k of a function $f : \mathbb{R} \to \mathbb{R}$ around some point $\bar{x}$ is given by:

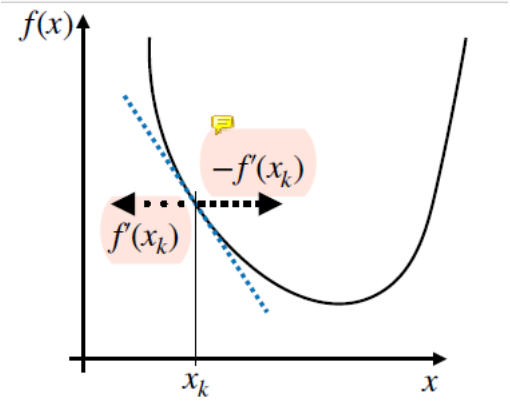
$$\sum_{n=0}^{k} \frac{f^n(\bar{x})}{n!}(x - \bar{x})^n$$

For a given x and in the limit $k \to +\infty$, the sum of the series may converge to the value $f(x)$ (e.g. this is the case for the Taylor expansion of e^x), or to something else, or diverge (e.g. the series of $\log(1 + x)$ for $x > 1$). Anyway it is fair to assume that increasing the order of the series we get better and better approximations for the first few orders as long as x is close enough to $\bar{x}$.

The first order expansion approximates f with a line that is tangent to f in the point $(\bar{x}, f(\bar{x}))$. The second order expansion approximates f with a parabola and so forth so on.

The derivative $f'(x^k)$ tells us the direction and the rate of the *local increase* of $f(x)$ around x^k. Therefore, during our procedure that seeks a minimizer, it seems convenient to choose the next candidate in the form $x^{k+1} = x^k + \alpha p^k$, where the step p^k is in the opposite direction to the one indicated by the derivative and where $\alpha > 0$ is a parameter that we call *step size* (also known as learning rate). Formally, assuming we are not already in a local minimum (where $f'(x^k) = 0$), we want to impose the *descent condition*

$$f'(x^k)p^k < 0, \tag{2.7}$$



Remember that in 1d derivative evaluated at a point is just some value and also update x, not f(x), parallel to x-axis.

From the two equations below:

$$f(x) = f(x^k) + f'(x^k)(x - x^k) + o(x - x^k).$$

and

$$x^{k+1} = x^k + \alpha p^k,$$

(the first equation holds for local moves, for x (x^{k+1}) and x^k close enough, thus for small alpha*p^k)

we get:

$$f(x^{k+1}) = f(x^k) + \alpha f'(x^k)p^k + o(\alpha p^k).$$

The descent condition:

$$f'(x^k)p^k < 0,$$

We want decrease so move in the opposite direction to the gradient. This guarantees that the cost will be reduced in the next iteration (as long as alpha is small enough – for a local move, so that the iterations are able to converge to a local minimum, but also large enough to save time).

Algorithm 1 Steepest Descent 1D

Require: cost function f, step size $\alpha > 0$, init. conf. x^0, max iters k_{MAX}
Ensure: candidate minimun of f
  $x \leftarrow x^0$
  for $k \leftarrow 1, ..., k_{MAX}$ do
    $x \leftarrow x - \alpha f'(x)$
  end for
  return x

Multiple dimensions

$$\nabla f(\boldsymbol{x}) = \left(\left.\frac{\partial f}{\partial x_1}\right|_x, \left.\frac{\partial f}{\partial x_2}\right|_x, \ldots, \left.\frac{\partial f}{\partial x_n}\right|_x \right).$$

$$\left.\frac{\partial f}{\partial x_i}\right|_x = \lim_{\epsilon \to 0} \frac{f(x_1, \ldots, x_i + \epsilon, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{\epsilon}$$

When the gradient is defined and continuous everywhere, the function is said to be continuously differentiable.

Symmetry and billinearity of scalar product.

$$\|u\| := \sqrt{\langle \boldsymbol{u}, \boldsymbol{u} \rangle} = \sqrt{\sum_{i=1}^{n} u_i^2}.$$

$$\langle \boldsymbol{u}, \boldsymbol{v} \rangle = \|\boldsymbol{u}\| \|\boldsymbol{v}\| \cos\theta,$$

First-order Taylor expansion around xbar:

$$\begin{aligned} f(\boldsymbol{x}) &= f(\overline{\boldsymbol{x}}) + \sum_{i=1}^{n} \left.\frac{\partial f}{\partial x_i}\right|_{\overline{\boldsymbol{x}}} (x_i - \overline{x}_i) + o(\|\boldsymbol{x} - \overline{\boldsymbol{x}}\|) \\ &= f(\overline{\boldsymbol{x}}) + \langle \nabla f(\overline{\boldsymbol{x}}),\, \boldsymbol{x} - \overline{\boldsymbol{x}} \rangle + o(\|\boldsymbol{x} - \overline{\boldsymbol{x}}\|) \end{aligned}$$

$\boldsymbol{x}^*$ is a local minimizer if $\exists \epsilon > 0$ such that for any x with $\|\boldsymbol{x} - \boldsymbol{x}^*\| < \epsilon$ we have:

$$f(\boldsymbol{x}) \geq f(\boldsymbol{x}^*). \tag{3.8}$$

Convex functions have only global minimizers!

Theorem 1. *First Order Necessary Condition*
Given a continuously differentiable function $f : \mathbb{R}^n \to \mathbb{R}$, any local minimizer x^* of f satisfies the "stationarity condition"

$$\nabla f(x^*) = 0 \tag{3.9}$$

Proof. If by contradiction we set $\nabla f(x^*) \neq 0$, then $\langle \nabla f(x^*), \nabla f(x^*) \rangle > 0$. Fix $\epsilon > 0$ and set $p = -\nabla f(x^*)$, then Eq. (3.7) gives

$$f(x^* + \epsilon p) = f(x^*) + \epsilon \langle \nabla f(x^*), p \rangle + o(\epsilon) \tag{3.10}$$

with $\langle \nabla f(x^*), p \rangle < 0$. As a consequence, $f(x^* + \epsilon p) < f(x^*)$ for ϵ small enough. In order to see this, consider the following. By definition of the small$-o$ notation, there is a $\bar{\epsilon} > 0$ such that for any ϵ, $0 < \epsilon < \bar{\epsilon}$, we have:

$$|o(\epsilon)| \leq -\frac{1}{2}\epsilon \langle \nabla f(x^*), p \rangle. \tag{3.11}$$

Therefore

$$f(x^* + \epsilon p) = f(x^*) + \epsilon \langle \nabla f(x^*), p \rangle + o(\epsilon) \tag{3.12}$$

$$\leq f(x^*) + \epsilon \langle \nabla f(x^*), p \rangle - \frac{1}{2}\epsilon \langle \nabla f(x^*), p \rangle \tag{3.13}$$

$$= f(x^*) - \frac{1}{2}\epsilon \langle \nabla f(x^*), p \rangle \tag{3.14}$$

which leads to $f(x^* + \epsilon p) < f(x^*)$. As a consequence x^* cannot be a local minimizer of f. □

The proof of the above theorem relies on the fact that, when we have a non zero gradient $\nabla f(x)$, it is always possible to choose a direction p such that we are sure to be able to decrease the cost if the step in that direction is small enough. Therefore, at any position x, we call *descent direction* a vector p that satisfies

$$\langle \nabla f(x), p \rangle < 0. \tag{3.15}$$

Consider a quadratic function in n dimensions:

$$f(x) = \frac{1}{2} \sum_{i,j=1}^{n} x_i A_{ij} x_j + \sum_{i=1}^{n} b_i x_i + c \tag{3.16}$$

Equivalent to:

$$f(x) = \frac{1}{2} x^T A x + b^T x$$

Matrix A positive definite: it is invertible (non-zero eigenvalues) and the function is convex and has a unique (global) minimum.

Gradient of such function is:

$$\nabla f(x) = Ax + b$$

We can find the minimum by imposing the stationarity condition $\nabla f(x^*) = 0$:

$$Ax^* + b = 0 \;\Rightarrow\; x^* = -A^{-1}b \tag{3.19}$$

The displacement which leads to the global minimum in one step in the quadratic case (which we computed as $-A^{-1}b$) is:

$$\begin{aligned} p &= -A^{-1}b - x \\ &= -A^{-1}(b + Ax) \end{aligned}$$

which in terms of Hessian and gradient is:

$$p = -\left(\nabla^2 f(x)\right)^{-1} \nabla f(x)$$
$$x^* = x + p$$

Therefore, starting from any position we can reach the minimum in one step if we choose the step as the right combination of the Hessian and the gradient.

If a function f can be differentiated twice, the *second order Taylor expansion* around a point $\overline{x}$ reads

$$f(x) = f(\overline{x}) + \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}(\overline{x})\,(x_i - \overline{x}_i) + \frac{1}{2}\sum_{i,j=1}^{n} \frac{\partial^2 f}{\partial x_i \partial x_j}(\overline{x})\,(x_i - \overline{x}_i)(x_j - \overline{x}_j) + o(\|x - \overline{x}\|^2) \tag{3.26}$$

$$= f(\overline{x}) + \langle \nabla f(\overline{x}),\, x - \overline{x}\rangle + \frac{1}{2}\langle \nabla^2 f(\overline{x})\,(x - \overline{x}),\, x - \overline{x}\rangle + o(\|x - \overline{x}\|^2) \tag{3.27}$$

or, using vector notation, we can write equivalently

$$f(x) = f(\overline{x}) + \nabla f(\overline{x})^T (x - \overline{x}) + \frac{1}{2}(x - \overline{x})^T \nabla^2 f(\overline{x})\,(x - \overline{x}) + o(\|x - \overline{x}\|^2) \tag{3.28}$$

$$f(\overline{x} + p) = f(\overline{x}) + \nabla f(\overline{x})^T p + \frac{1}{2} p^T \nabla^2 f(\overline{x})\, p + o(\|p\|^2)$$

Theorem 2. *Second Order Necessary Condition*

Given a twice continuously differentiable function $f : \mathbb{R}^n \to \mathbb{R}$, for any local minimizer x^* of f, we have that $\nabla f(x^*) = 0$ and that $\nabla^2 f(x^*)$ is positive semi-definite.

Proof. By Theorem 1 we have $\nabla f(x^*) = 0$. If by contradiction we assume $\nabla^2 f(x^*)$ not positive semi-definite, then there is a $p \neq 0$ such that $p^T \nabla^2 f(x^*) p < 0$. Fix $\epsilon > 0$, then Eq. (3.28) gives

$$f(x^* + \epsilon p) = f(x^*) + \epsilon^2 \frac{1}{2} p^T \nabla^2 f(x^*) p + o(\epsilon^2). \tag{3.30}$$

As a consequence, $f(x^* + \epsilon p) < f(x^*)$ for ϵ small enough, therefore x^* cannot be a local minimizer of f. □

Using the Taylor expansion of f it is easy to see that the gradient has to be orthogonal to a contour line. In fact if $x = (x_1, x_2)$ is on a contour line, and $x + p$ is another point on it, by definition of the contour line we have $f(x) = f(x + p)$. But the Taylor expansion also gives:

$$f(x + p) = f(x) + \langle \nabla f(x), p \rangle + o(\|p\|) \tag{3.31}$$

which implies that if we decrease $\|p\|$ while staying on the contour line

$$\langle \nabla f(x), \frac{p}{\|p\|} \rangle = o(1) \tag{3.32}$$

Therefore, since $p/\|p\|$ becomes the unit vector tangent to the contour line, we proved that $\nabla f(x)$ has to be orthogonal to the contour line. This argument can be easily generalized to n dimensional spaces, where equal $f : \mathbb{R}^n \to \mathbb{R}$ contours are hypersurfaces of dimension $n - 1$ and the gradient computed in a point on one of the hypersurfaces is orthogonal to the hypersurface itself.

Staying on the contour line implies f(x+p)=f(x) and given the formula for f(x+p), we want the second term to be 0 so that f(x+p)=f(x) is fulfilled, that happens when the scalar product of p and gradient is 0. Moving along the contour line implies moving in the direction tangent to the contour line so p is tangent to contour lines and for the scalar product of the tangent and the gradient to be 0, gradient must be orthogonal to the tangent to the contour line at a given point, hence it must be perpendicular to the contour line itself at that point.

Newton method

Using second-order information:

1 dimension

$$f(x) = f(x^k) + f'(x^k)(x - x^k) + \frac{1}{2} f''(x^k)(x - x^k)^2 + o((x - x^k)^2), \tag{2.9}$$

$$f'(x) = f'(x^k) + f''(x^k)(x - x^k) + o(x - x^k). \tag{2.10}$$

Multiple dimensions

Hessian matrix is the generalization to many dimension of the second derivative.

$$(\nabla^2 f(x))_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}(x)$$

The Hessian is a symmetric matrix by Schwarz's theorem (under some weak assumptions).

$$\nabla^2 f(x) = A$$

Line search methods

cost being an exception). The first order Taylor expansion around a certain x:

$$f(x+p) = f(x) + \langle \nabla f(x), p \rangle + o(\|p\|), \tag{3.34}$$

tells us that as long as the gradient is non-zero and we select a displacement p *small enough* and *partially anti-aligned* to the gradient, we can decrease the value of our objective function going from x to $x+p$. Therefore it is convenient to introduce the notion of descent direction. A *descent direction* p at position x is a direction which is partially anti-aligned to the gradient, that is

$$\langle p, \nabla f(x) \rangle < 0. \tag{3.35}$$

It is conceptually useful to decompose the displacement vector $\Delta^k = x^{k+1} - x^k$ in two parts, $\Delta^k = \alpha^k p^k$, where the vector p^k is a descent direction (which doesn't have to be a unit vector though) and $\alpha^k \geq 0$ is a scalar called step size or learning rate.

Algorithm 2 General Line Search Procedure

$x \leftarrow x^0$
for $k \leftarrow 1, \ldots, k_{MAX}$ **do**
  Choose descent direction p^k
  Choose step-size α^k
  $x \leftarrow x + \alpha^k p^k$
end for
return x

In order to have convergence guarantees and to achieve optimal speed, the rule for choosing the step-size should be carefully tuned based on the method used for choosing the direction p^k (different trade-offs in terms of computational complexity and convergence rate). A learning rate too small would be too slow to find the solution, while a learning rate too large would cause big fluctuations and never converge to the solution.

In general (choice of B_k varies from one method to another):

$$p^k = -B_k^{-1} \nabla f_k.$$

1. Steepest (gradient) descent: B_k=I, so the update rule is:

$$x^{k+1} = x^k - \alpha^k \nabla f_k$$

2. Newton method (computationally expensive to compute n^2 second derivatives at each step):

Newton Method. The Newton method is provably (in strictly convex settings) faster in terms of number of iterations than gradient descent since it uses the second order information contained in the Hessian. For this method we have $B_k = \nabla^2 f_k$, therefore

$$p^k = -\left(\nabla^2 f_k\right)^{-1} \nabla f_k \tag{3.39}$$

3. Quasi-Newton methods:

Quasi-Newton methods. In Quasi-Newton methods one tries to compute an approximation B_k as close as possible to the true Hessian $\nabla^2 f_k$ without computing second order derivatives but using first order information from the past trajectory. According to these methods, at each step k of the algorithm there is a quadratic model that is supposed to be a good approximation of the function $f(x)$ in a neighborhood of x^k. Of course, the best possible local quadratic approximation is the second Taylor expansion of $f(x)$ at x^k, but since we don't want to compute the Hessian, we will have some matrix B_k ideally approximating it. In this way our local model, that we call m_k, is roughly close to the Taylor expansion. Therefore, we assume that at step k the following approximation of f holds for some (possibly small) displacement p:

$$f(x^k + p) \approx m_k(p) = f_k + \nabla f_k^T p + \frac{1}{2} p^T B_k p.$$

We see that the only term that we have to fix in our local model is the "quasi-Hessian" B_k. This is done by starting with some guess B_0 at time 0 (typically $B_0 = I$) and then iteratively updating B_k according to some algorithmic specific rules. Once we have the model $m_k(p)$, we select the descent direction by doing exact minimization on the model (we can do it since it is quadratic). This leads to a new position $x^{k+1} = x^k + \alpha^k p^k$ once also the step size is considered (usually we set $\alpha^k < 1$ since we are not fully confident in our local model). Once we have x^{k+1}, we have to determine the new local model m_{k+1}, again in the form:

$$f(x^{k+1} + p) \approx m_{k+1}(p) = f_{k+1} + \nabla f_{k+1}^T p + \frac{1}{2} p^T B_{k+1} p. \tag{3.40}$$

According to some criterion the Hessian approximation at the new position is computed as an update to the old Hessian approximation:

$$B_{k+1} = B_k + U_k. \tag{3.41}$$

The update matrix U_k is often chosen as a low rank matrix: rank 1 for the symmetric update we shall discuss in Section 3.9, and rank 2 for the BFGS method, one of the most robust and most commonly used quasi-Newton methods.

Among the quasi-Newton methods, one of the simplest is the Symmetric Rank-1 update rule. It constructs an approximation B_k of the true Hessian $\nabla^2 f_k$ using only first order information as follows:

1. Choose some $\boldsymbol{v} \in \mathbb{R}^n$ and $\sigma \in \{-1, 1\}$ optimal for the problem ;
2. $B_0 = I$;
3. At each step, update B with a rank-1 perturbation: $B_{k+1} = B_k + \sigma \boldsymbol{v} \cdot \boldsymbol{v}^T$. Note that $\boldsymbol{v} \cdot \boldsymbol{v}^T$ is a $n \times n$ matrix with rank 1.

In particular, suppose to construct a local model m_{k+1} such that

$$f(\boldsymbol{x}^{k+1} + \boldsymbol{p}) \approx m_{k+1}(\boldsymbol{p}) = f(\boldsymbol{x}^{k+1}) + (\nabla f(\boldsymbol{x}^{k+1}))^T \boldsymbol{p} + \frac{1}{2} \boldsymbol{p}^T B_{k+1} \boldsymbol{p} \tag{3.43}$$

For $m_{k+1}(\boldsymbol{p})$ to be a good approximation in the neighborhood of $\boldsymbol{x}^{k+1}$, we impose the following requirements:

1. $f(\boldsymbol{x}^{k+1}) = m_{k+1}(\boldsymbol{0})$
2. $\nabla f(\boldsymbol{x}^{k+1}) = \nabla m_{k+1}(\boldsymbol{0})$
3. $\nabla m_{k+1}(-\alpha_k \boldsymbol{p}^k) = \nabla f(\boldsymbol{x}^k)$

Gradient of m_k+1, we consider gradient of gradient of f evaluated at -alpha_k*pk to be gradient of m_k_1 evaluated at –alpha*p^k, which then get's replaced by B_k.

While the first two conditions are met by construction by the class of models in Eq. (3.43), the last one constrains B_{k+1} as follows:

$$\begin{aligned}\nabla m_{k+1}(-\alpha_k \boldsymbol{p}^k) &= \nabla f(\boldsymbol{x}^{k+1}) + B_{k+1}(-\alpha_k \boldsymbol{p}^k) = \nabla f(\boldsymbol{x}^k) \\ &\Rightarrow \nabla f(\boldsymbol{x}^{k+1}) - \nabla f(\boldsymbol{x}^k) = B_{k+1}(\alpha_k \boldsymbol{p}^k)\end{aligned}$$

Let $\nabla f(\boldsymbol{x}^{k+1}) - \nabla f(\boldsymbol{x}^k) = \boldsymbol{y}^k$ and $\alpha_k \boldsymbol{p}^k = \boldsymbol{s}^k$, then

$$\begin{aligned}\boldsymbol{y}^k = B_{k+1}\boldsymbol{s}^k = (B_k + \sigma \boldsymbol{v}\boldsymbol{v}^T)\boldsymbol{s}^k = B_k\boldsymbol{s}^k + \sigma \boldsymbol{v}^T\boldsymbol{s}^k\boldsymbol{v} \\ \Rightarrow \boldsymbol{y}^k - B_k\boldsymbol{s}^k = \sigma \boldsymbol{v}^T\boldsymbol{s}^k\boldsymbol{v}\end{aligned}$$

Let $\boldsymbol{v} = \delta(\boldsymbol{y}^k - B_k\boldsymbol{s}^k)$, then

$$\begin{aligned}\boldsymbol{y}^k - B_k\boldsymbol{s}^k = \sigma\delta^2(\boldsymbol{y}^k - B_k\boldsymbol{s}^k)^T\boldsymbol{s}^k(\boldsymbol{y}^k - B_k\boldsymbol{s}^k) \\ \Rightarrow \sigma\delta^2(\boldsymbol{y}^k - B_k\boldsymbol{s}^k)^T\boldsymbol{s}^k = 1\end{aligned}$$

which leads to

$$\Rightarrow \begin{cases}\delta = |(\boldsymbol{y}^k - B_k\boldsymbol{s}^k)^T\boldsymbol{s}^k|^{-\frac{1}{2}} \\ \sigma = \text{sign}((\boldsymbol{y}^k - B_k\boldsymbol{s}^k)^T\boldsymbol{s}^k) \\ B_{k+1} = B_k + \sigma\boldsymbol{v}\boldsymbol{v}^T\end{cases}$$

and finally to

$$B_{k+1} = B_k + \sigma\frac{(\boldsymbol{y}^k - B_k\boldsymbol{s}^k)(\boldsymbol{y}^k - B_k\boldsymbol{s}^k)^T}{(\boldsymbol{y}^k - B_k\boldsymbol{s}^k)^T\boldsymbol{s}^k}$$

Makes sense

Learning rate:

Ideally we want the step size close to:

$$\alpha^k = \arg\min_{\alpha \geq 0} f(x^k + \alpha p^k).$$

- **Fixed step size.** $\alpha^k = \alpha \quad \forall\, k$ (see Fig. 3.2).
- **Decreasing step size.** E.g. $\alpha^k = \frac{\alpha^0}{k}$ or some other decreasing function of the step k
- **Dependent on some conditions.** There are some (complicated) adaptive

Momentum methods

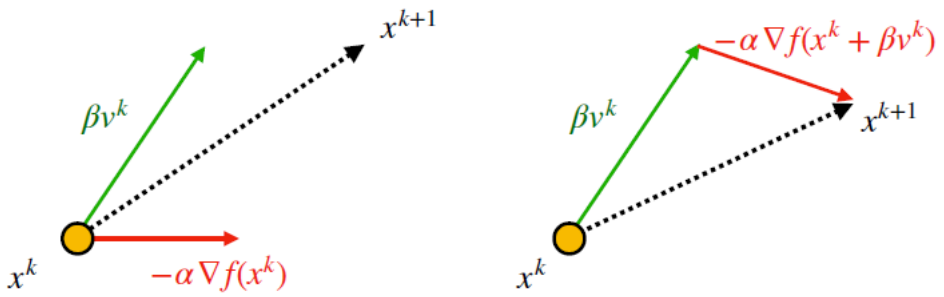


Figure 3.3: Standard Momentum compared to Nesterov Momentum.

The general strategy used in momentum methods is to add an inertial term, called *momentum* (in physics momentum is defined as mass $\times$ velocity). This momentum term makes it possible that in almost flat regions, where the gradient is very small but the minimum is still far away, the iteration keeps taking large steps due to the velocity acquired during the trajectory. This is a little departure from the general line search departure, since the direction taken in momentum methods is not necessarily a descent one. Momentum methods come in many flavors.

- **Standard Momentum.** In classical mechanics, the motion of a body of mass m subject to a force $\boldsymbol{F}$ is given by Newton's second law:

$$\begin{cases} m\frac{d\boldsymbol{v}}{dt} = \boldsymbol{F} \\ \frac{d\boldsymbol{x}}{dt} = \boldsymbol{v} \end{cases}$$

That is, a force produces a change in velocity and a velocity produces a change in position. By analogy between the force $\boldsymbol{F}$ and the gradient of our cost function, we can adapt the natural law of motion to our discrete time iterative procedure as follows. Let's call $\beta \in [0,1]$ a parameter which is associated to inertia. A popular choice is $\beta = 0.9$. We start with a velocity $v^k = \mathbf{0}$ and in following steps we apply the recursion rule

$$\begin{cases} v^{k+1} = \beta v^k - \alpha^k \nabla f_k \\ x^{k+1} = x^k + v^k \end{cases}$$

- **Nesterov Momentum.** Nesterov Momentum is just a small variation of the standard momentum that often proves to be convenient in terms of performance. The difference is in the fact that now the velocity is calculated based on the gradient computed at the current position shifted to the approximated direction towards which we are headed (see Fig. (3.3)).

$$\begin{cases} v^{k+1} = \beta v^k - \eta \nabla f(x^k + \beta v^k) \\ x^{k+1} = x^k + v^k \end{cases}$$

  Nesterov's update rule can take different forms. Also, in optimization literature, it is known as accelerated gradient method or fast gradient method.

- **Adam, AdaGrad, RmsProp, ...** There are many heuristic optimization methods that, while incorporating the momentum idea, try also to compute a local approximation of the local curvature (the Hessian), as in quasi-Newton methods, and to average out noisy fluctuations of the gradient. In these methods, the approximation to the Hessian always takes a diagonal form. See `http://ruder.io/optimizing-gradient-descent/` for a comparison of these methods.

Convex functions

1) each minimizer is a global minimizer; 2) along any direction, the slope (i.e. the directional derivative) is a monotonous function, either increasing or decreasing. Moreover, for strictly convex function, there is a unique minimum.

A *convex set* $S \subseteq \mathbb{R}^n$ is a set such that for any two points $x, y \in S$ the whole segment with extremities in x and y is contained in S, that is for any $c \in [0,1]$ we have

$$cx + (1-c)y \in S. \tag{A.1}$$

A *convex function* f is a function defined on a convex set S and such that for any two points $x, y \in S$ the segment connecting the points $(x, f(x))$ and the point $(y, f(y))$ stays above the graph of the function, that is for any $c \in [0,1]$ we have

$$cf(x) + (1-c)f(y) \geq f(cx + (1-c)y). \tag{A.2}$$

Convergence rate

The performance of a linear search implementation is defined in terms of convergence rate. Given a succession $\{x^k\}_k$ with limit x^*, we say that the convergence of the sequence is q-linear, for some $q \geq 1$, if for some $c \in (0,1)$ and for some $k_0 > 0$ we have

$$||x^{k+1} - x^k|| \leq c||x^* - x^k||^q \qquad \text{for } k \geq k_0. \tag{B.1}$$

In particular, for different choices of descent direction p^k and for some appropriate step sizes α^k, there are different convergence rates for the algorithm:

- Gradient descent: $q = 1$ → linear rate
- Quasi-Newton methods: $1 \leq q \leq 2$ → super-linear rate
- Newton method: $q = 2$ → quadratic rate

It is easy to convince yourself, by iterative application of Eq. (B.1), that for the linear rate ($q = 1$) we have

$$\|x^* - x^k\| \leq Ac^k \tag{B.2}$$

so that the distance from the minimum is exponentially shrinking in time. Super linear rates have even faster convergence.

Update:

Gradient-descent

$$x^{k+1} = x^k - \alpha^k \nabla f_k$$

Newton

$$p^k = -(\nabla^2 f_k)^{-1} \nabla f_k$$

$$p = -(\nabla^2 f(x))^{-1} \nabla f(x)$$
$$x^* = x + p$$

Finite differences

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

$$f(x+h) = f(x) + hf'(x) + h^2 \frac{f''(\xi)}{2!} \quad \text{where } \xi \text{ is some number between } x \text{ and } x+h.$$

Instead of little o notation^.

$$\frac{f(x+h)-f(x)}{h} - f'(x) = h\frac{f''(\xi)}{2},$$

which tells us that the error is proportional to h to the power 1, so $\frac{f(x+h)-f(x)}{h}$ is said to be a "first-order" approximation.

If $h > 0$, say $h = \Delta x$ where Δx is a finite (as opposed to infinitesimal) positive number, then

$$\frac{f(x+\Delta x)-f(x)}{\Delta x}$$

is called the first-order or $O(\Delta x)$ *forward difference* approximation of $f'(x)$.
If $h < 0$, say $h = -\Delta x$ where $\Delta x > 0$, then

$$\frac{f(x+h)-f(x)}{h} = \frac{f(x)-f(x-\Delta x)}{\Delta x}$$

is called the first-order or $O(\Delta x)$ *backward difference* approximation of $f'(x)$.

$$f(x+\Delta x) = f(x) + \Delta x\, f'(x) + \Delta x^2\frac{f''(x)}{2!} + \Delta x^3\frac{f'''(\xi_1)}{3!}, \qquad \xi_1 \in (x,\, x+\Delta x)$$
$$f(x-\Delta x) = f(x) - \Delta x\, f'(x) + \Delta x^2\frac{f''(x)}{2!} - \Delta x^3\frac{f'''(\xi_2)}{3!}, \qquad \xi_2 \in (x-\Delta x,\, x)$$

gives $f(x+\Delta x) - f(x-\Delta x) = 2\Delta x\, f'(x) + \Delta x^3\frac{(f'''(\xi_1)+f'''(\xi_2))}{6}$, so that

$$\frac{f(x+\Delta x)-f(x-\Delta x)}{2\Delta x} - f'(x) = \Delta x^2\frac{(f'''(\xi_1)+f'''(\xi_2))}{12}$$

Hence $\frac{f(x+\Delta x)-f(x-\Delta x)}{2\Delta x}$ is an approximation of $f'(x)$ whose error is proportional to Δx^2. It is called the second-order or $O(\Delta x^2)$ *centered difference* approximation of $f'(x)$.

$$f(x+\Delta x) = f(x) + \Delta x\, f'(x) + \Delta x^2\frac{f''(x)}{2!} + \Delta x^3\frac{f'''(x)}{3!} + \Delta x^4\frac{f^{(4)}(\xi_1)}{4!}\cdots$$
$$f(x-\Delta x) = f(x) - \Delta x\, f'(x) + \Delta x^2\frac{f''(x)}{2!} - \Delta x^3\frac{f'''(x)}{3!} + \Delta x^4\frac{f^{(4)}(\xi_2)}{4!}\cdots$$

gives $f(x+\Delta x) + f(x-\Delta x) = 2f(x) + \Delta x^2 f''(x) + \Delta x^4\frac{(f^{(4)}(\xi_1)+f^{(4)}(\xi_2))}{24}$, so that

$$\frac{f(x+\Delta x)-2f(x)+f(x-\Delta x)}{\Delta x^2} - f''(x) = \Delta x^2\frac{(f^{(4)}(\xi_1)+f^{(4)}(\xi_2))}{24}$$

Hence $\frac{f(x+\Delta x)-2f(x)+f(x-\Delta x)}{\Delta x^2}$ is a second-order centered difference approximation of the second derivative $f''(x)$.

$O(\Delta x^2)$ centered difference approximations:

$$f'(x): \quad \{f(x+\Delta x) - f(x-\Delta x)\}/(2\Delta x)$$
$$f''(x): \quad \{f(x+\Delta x) - 2f(x) + f(x-\Delta x)\}/\Delta x^2$$

$O(\Delta x^2)$ forward difference approximations:

$$f'(x): \quad \{-3f(x) + 4f(x+\Delta x) - f(x+2\Delta x)\}/(2\Delta x)$$
$$f''(x): \quad \{2f(x) - 5f(x+\Delta x) + 4f(x+2\Delta x) - f(x+3\Delta x)\}/\Delta x^3$$

$O(\Delta x^2)$ backward difference approximations:

$$f'(x): \quad \{3f(x) - 4f(x-\Delta x) + f(x-2\Delta x)\}/(2\Delta x)$$
$$f''(x): \quad \{2f(x) - 5f(x-\Delta x) + 4f(x-2\Delta x) - f(x-3\Delta x)\}/\Delta x^3$$

$O(\Delta x^4)$ centered difference approximations:

$$f'(x): \quad \{-f(x+2\Delta x) + 8f(x+\Delta x) - 8f(x-\Delta x) + f(x-2\Delta x)\}/(12\Delta x)$$
$$f''(x): \quad \{-f(x+2\Delta x) + 16f(x+\Delta x) - 30f(t) + 16f(x-\Delta x) - f(x-2\Delta x)\}/(12\Delta x^2)$$

It is appropriate to use a forward difference at the left endpoint x = x1, a backward difference at the right endpoint x = xn, and centered difference formulas for the interior points.

Data structures

Optimizing an algorithm also includes choosing the most appropriate data structure. Trade-off: memory usage vs computational complexity.

Optimizing find-max/ find-min, predecessor/ successor at the expense of indexing/ assignment and insertion/ deletion.

Lists: indexing/ assignment O(1), insertion/ deletion O(n), max/min/successor/predecessor O(n).

Dictionary: indexing/ assignment O(1), insertion/ deletion O(1) (on average; hash table), findmax/ successor O(n). But possible collision between the time and space complexity.

Binary search trees: indexing/assignment and insert/deleta O(log n), findmax/ successor O(log n) (on average). Log n is the height of the tree, while traversing, we do binary decisions. A lot more structure, which the algorithm can exploit to perform operations.

Tree: undirected, acyclic, connected graph.

Binary tree (recursive definition): a structure defined on a finite set of nodes (vertices of a tree). It is either:

1. Empty
2. Composed of 3 disjoint sets of nodes:

- Root
- Left sub-tree (still a BT)
- Right sub-tree (still a BT)

Parts: root (no parent), internal nodes and leaves (no children).

Quantities to be specified for each node (attributes of a node):

- Identifier – “key”
- Parent
- Left (child)
- Right (child)

To connect 2 nodes, a and b, update the parent of b and a child of a.

Relationship between the height (depth) of a binary tree and the number of nodes it contains. For a complete tree (for which all slots in previous levels are filled before starting to populate the next level), there is a logarithmic increase in depth with the number of nodes. Speed gain from O(n) to O(log n) if there is ordering. Worst case: h=n, only right children.

A binary search tree is a binary tree + a strong ordering property (on the keys; and a layered structure). BST property:

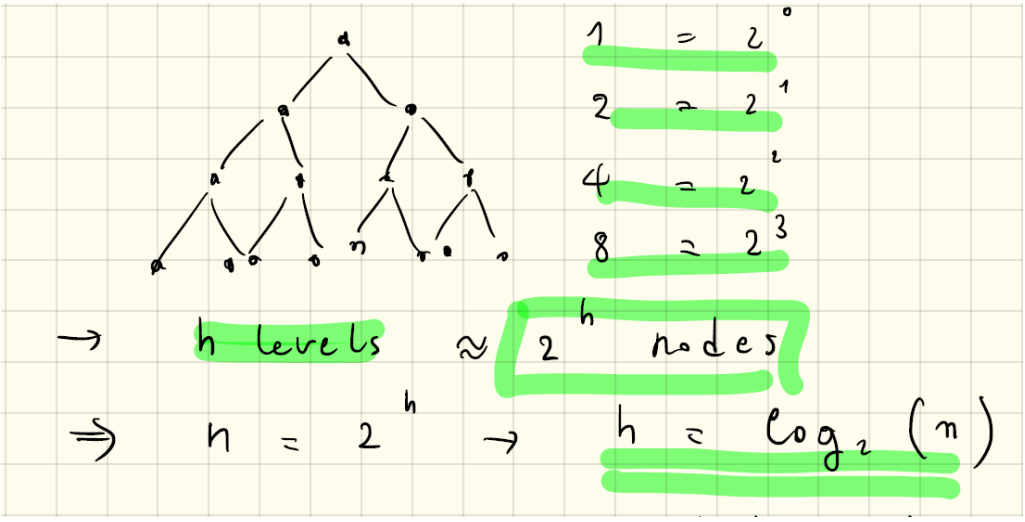
For every node x in the BST:

- For every node y in the left-subtree of x: x.key>y.key;
- For every node y in the right-subtree of x: x.key<y.key.
- We are using strict comparison because we want to possibly use the BST as a dictionary (keys cannot be repeated).

Organizing a random vector of integers as a BST:

a) Base case: tree is empty – set the first element as the root of the tree;
b) Compare the new element with the root: if it is smaller, put it on the left, if greater, put it on the right.
c) Keep repeating the procedure b) until you find the right sport for the element. Find a parent node that has the correct order relation and is missing the corresponding child. If key is already present, refuse to insert the node.

This process takes O(log n) for each entry and O(n log(n)) for the full BST.

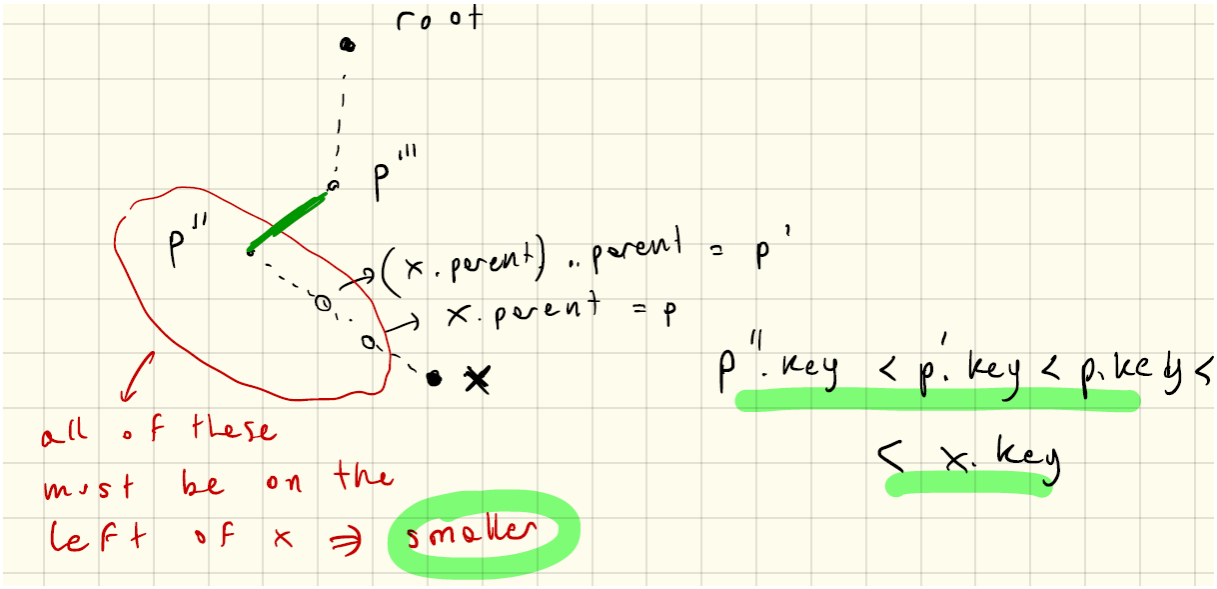


Tree-search (“indexing”): no linear structure so we have to look for the slot where that key would be and return it if found or return None otherwise.

Max and min: for max go right until the node has no right child and for min go left until the node has no left child.

Successor:

1. The node x which we want to compute the successor y of has the right child: y=min_subtree(r), where r=x.right.
2. The node which we want to compute the successor of does not have the right child. We move up the tree until we find the first parent to the right (the first parent which has the left child so which has the left subtree whose root is its left child – x is on this subtree). Below p'''.key>x.key, while for all other parents p''.key<p'.key<p.key<x.key.



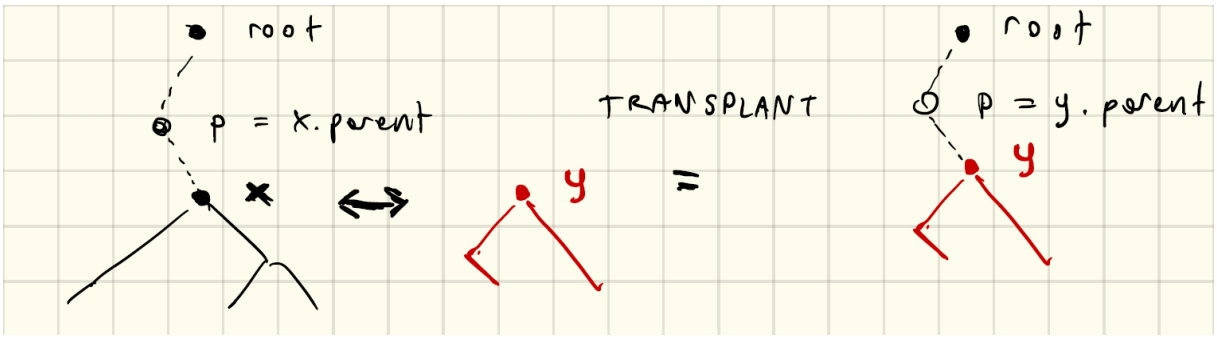
Predecessor:

1. Node x has the left child. y=max_subtree(l), where l=x.left.
2. Node x does not have the left child, then y is the first parent to the left (p.key<x.key) so when x is its right child.

DELETION

Transplant

A procedure that removes a subtree (from its root) and attaches a different subtree in its place.



Assuming that this is allowed (we need to make sure) i.e. y.key has the same relationship with x.parent as x.key. Y could be None, then the transplant operation would be just deleting the subtree rooted in x from the tree.

2 operations:

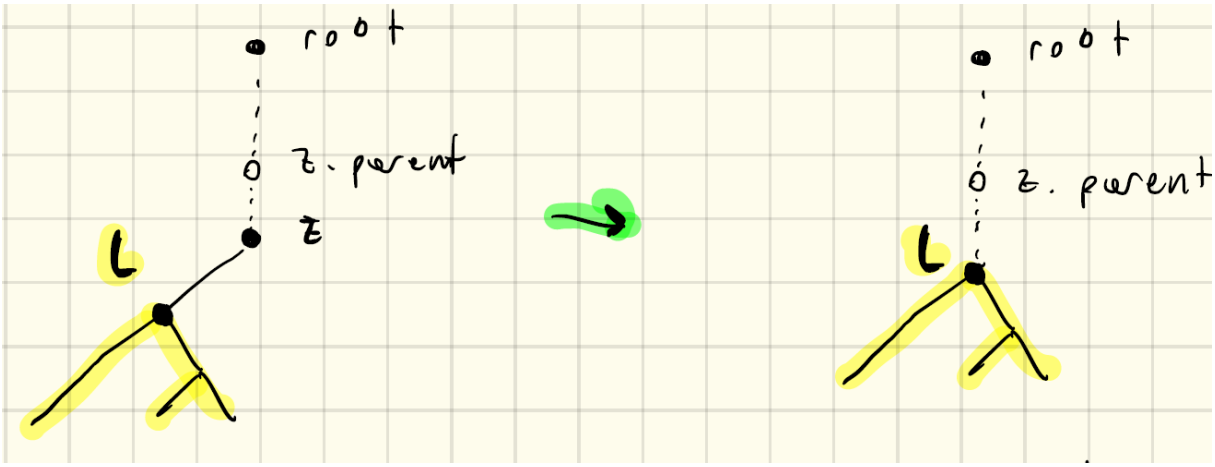
1. Update y.parent=x.parent
2. Update x.parent.right (or x.parent.left)=y (depending whether x was a right or left child)

Deletion

Notation: z is the node to be removed, r is its potential right child, l is its potential left child, y is its potential successor, x is y's potential right child.

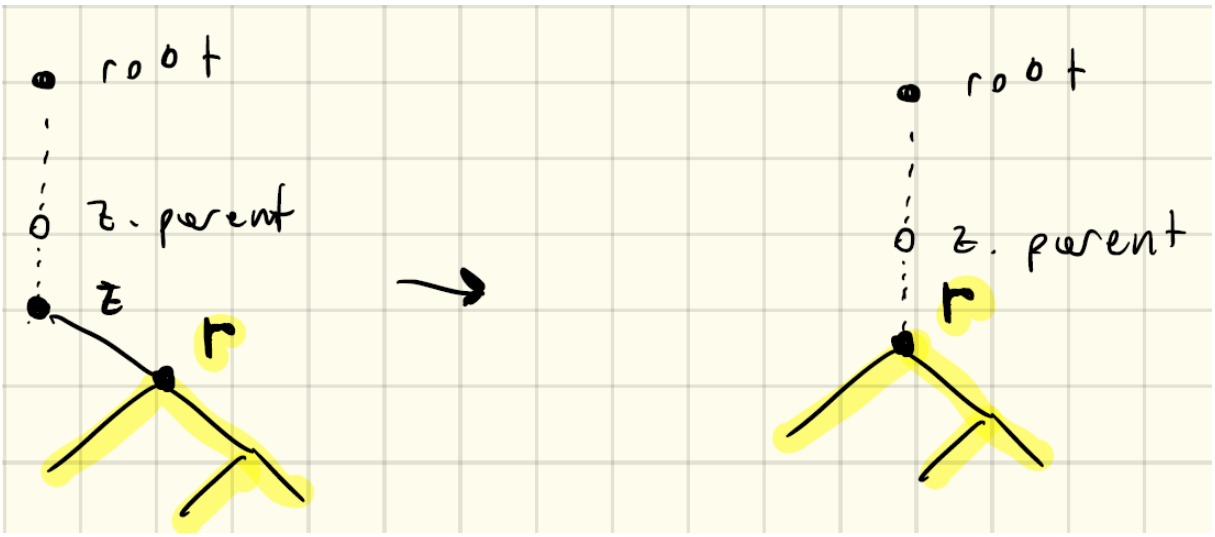
The best replacement for z is its successor if it exists, i.e. if z has the right subtree (nodes on the left sub-tree are out of consideration as long as z has the right subtree because the property of BST says that x.key>x.left.key). Useful property: the successor does not have the left subtree (the left child) so we can freely attach the left subtree of former z to the node which replaced z. None stands for non-existence.

1. z has no right child



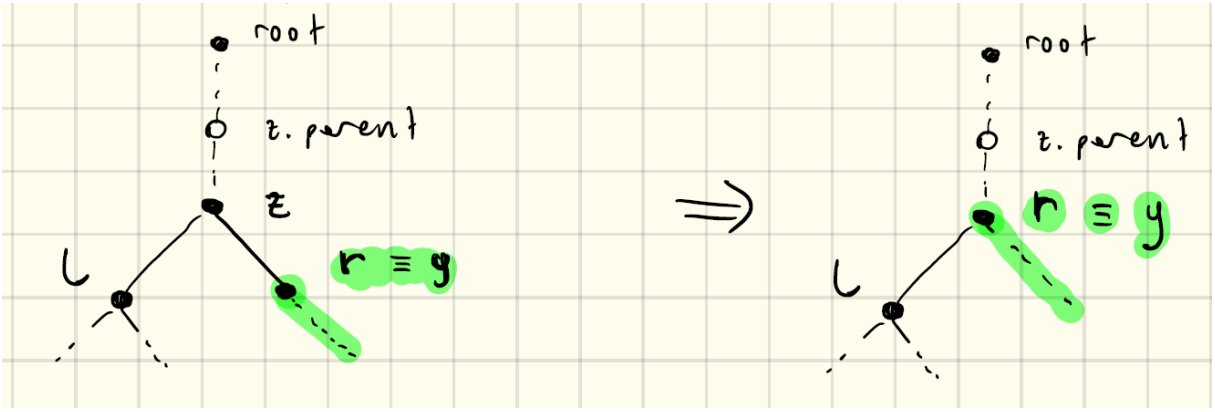
Transplant z→l (l and its subtree in the place of z) (also works if l is none)

2. z has no left child



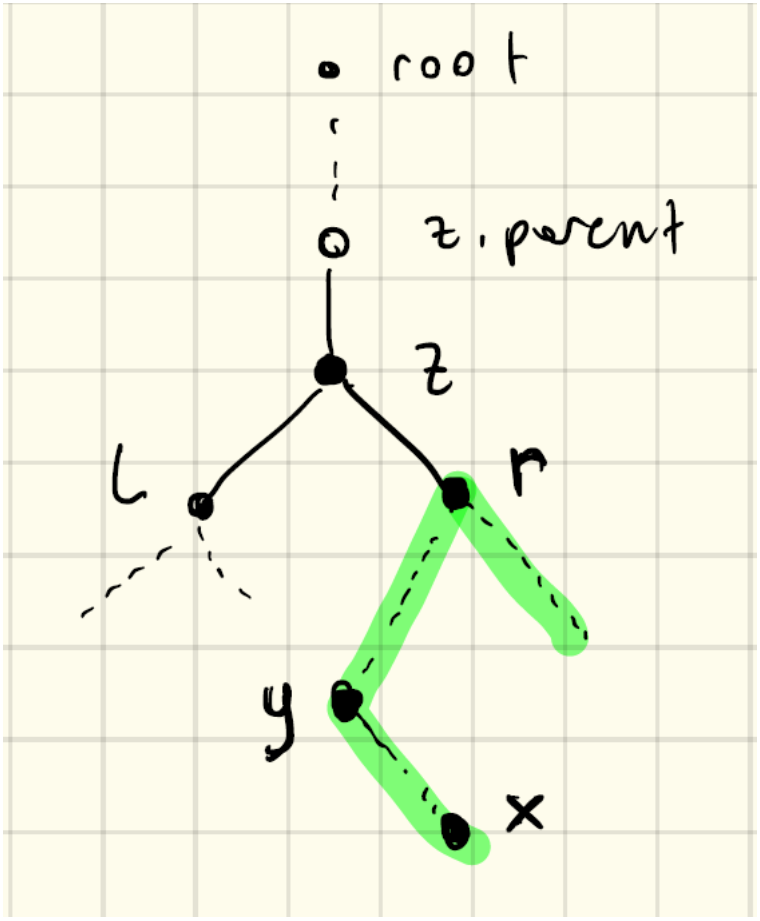
Transplant z→r (r and its subtree in its place) (also works if r is None).

3. y (the successor) is the right child of z

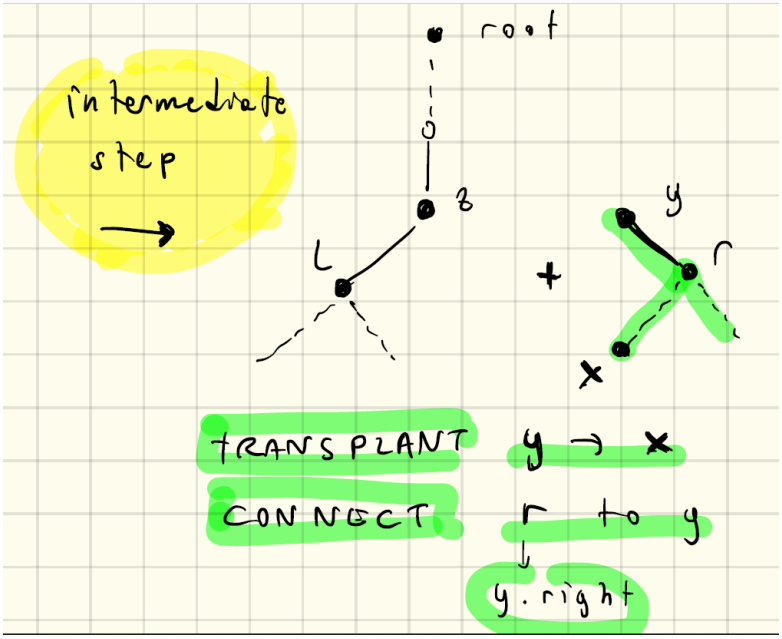


Transplant z→r (r in place of z) and connect l to r.

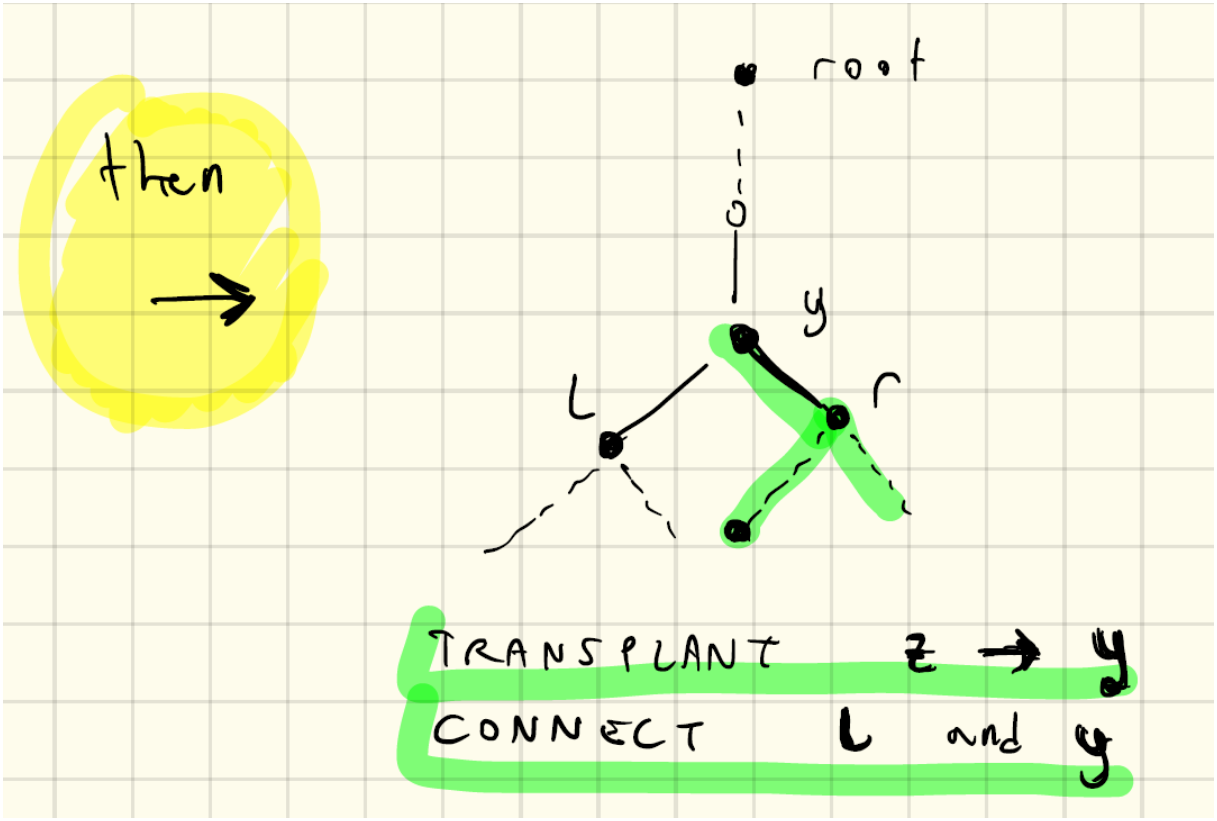
4. y (the successor) is not the right child of z



Relationships between nodes: y is the successor (by BST property, it does not have the left sub-tree, it may have the right-subtree, either empty or not). The right child of z may have both left and right sub-trees in all generality; the same goes for l. We have l.key<z.key<y.key<x.key<r.key so we want to replace z with y, then make y.left=l, y.right=r, r.left=x. (we do not want to make y.right=x and x.right=r because x may have right child already in general and we do not want to overcomplicate). Then we have a guarantee relationships implied by BST property are preserved.



Transplant y→x is replacing the subtree whose root is y with the subtree whose root is x (before y was the left child of r so now x becomes the left child of r). Connect r to y: making r the right child of y.



Replace z-subtree (subtree whose root is z) with y-subtree and connect l-subtree of former z with the placeholder of z, with the node y.

$$f(\hat{x},\hat{y}) = -\frac{1}{b}\sum_{i=0}^{n-1}\log\left[\sum_{j=0}^{k-1}\exp\left(-b\left((\hat{x}_j - x_i)^2 + (\hat{y}_j - y_i)^2\right)\right)\right]$$

If we write this function this way: c=-(1/b)*((np.log((np.exp(-b*((xc-np.reshape(xp, (n, 1)))**2+(yc-np.reshape(yp, (n, 1)))**2))).sum(axis=1))).sum()), we want row sums, also it makes sense since in the interior sum, xj_hat is variable and it is variable in rows according to our choice (we did xc and reshaped xp) and i is fixed for all j, then i+1 for another round over all j…

d = (xc - xp.reshape(n,1))**2 + (yc - yp.reshape(n,1))**2. Xc gathers x_hat and it is a row vector, first we want to sum for a given values of j, internal.

We want to do row sums first, we specify axis=1.

1. Consider the quadratic function $f : \mathbb{R}^n \to \mathbb{R}$ to be minimized defined by:

$$f(x) = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} x_i A_{ij} x_j + \sum_{i=1}^{n} b_i x_i = \frac{1}{2}x^T Ax + b^T x$$

   - We have

$$\frac{\partial f\,(x)}{\partial x_i} = \sum_j A_{ij}x_j + b_i$$

     This can be easily seen by unrolling the summations. E.g. for `n=2` and assuming A symmetric, we have

$$\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} x_i A_{ij} x_j = \frac{1}{2}A_{11}x_1x_1 + \frac{1}{2}A_{22}x_2x_2 + A_{12}x_1x_2$$

$$\sum_{i=1}^{n} b_i x_i = b_1x_1 + b_2x_2$$

     One can then easily computer the partial derivatives $\frac{\partial f(x)}{\partial x_i}$.

     The result for the partial derivative can be compactly written in an expression for the whole gradient:

$$\nabla f(x) = Ax + b$$

   - We apply the gradient descent rule

$$x' = x - \alpha \nabla f\,(x)$$

     to obtain

$$x' = x - 0.5\,(Ax + b)$$

PAGE RANK

2. The PageRank algorithm can be understood as a Markov chain on a directed graph of N web pages with transition matrix $Q_{ij} = A_{ij}/k_j$, where A is an adjacency matrix:

$$A_{ij} = \begin{cases} 1 & \text{if a link from webpage } j \text{ to webpage } i \text{ exists} \\ 0 & \text{otherwise} \end{cases}$$

   and $k_j = \sum_{i=1}^{N} A_{ij}$ is the out-degree of vertex j.

   In the very peculiar case in which the matrix A_{ij} is symmetric, and with the further assumption that the graph is connected, it is easy to guess the stationary distribution. With these assumptions, solve the following exercises:

   - Find the stationary distribution ρ and prove its stationarity.

ANSWERS:

- We can try to guess the stationary distribution looking at the detailed balance condition $Q_{ij}\rho_j = Q_{ji}\rho_i$, that in this case reads

$$\frac{A_{ij}}{k_j}\rho_j = \frac{A_{ji}}{k_i}\rho_i$$

Since A_{ij} is symmetric, it is easy to realize that if we take ρ_i proportional to the degree k_i, that is $\rho_i = k_i/Z$, the last equation is satisfied. Therefore we have

$$\rho_i = \frac{k_i}{\sum_{i'} k_{i'}} \qquad \forall i$$

Since the detailed balance condition is a sufficient condition for stationarity, we have showed that ρ as expressed in last equation is stationary.

This form for the stationary distribution is also reasonable considering that the PageRank algorithm aims at giving more relevance to web pages that have higher connectivity.

- Introduce a parameter $\beta \geq 0$ and consider a generalized page rank algorithm with transition matrix $Q^{(\beta)}$ derived from the previous one as follows:

$$Q_{ij}^{(\beta)} = Q_{ij} \min\left(1, \frac{k_i^\beta}{k_j^\beta}\frac{Q_{ji}}{Q_{ij}}\right) \qquad \forall i, j \text{ with i} \neq j.$$

In the case $Q_{ij} = Q_{ji} = 0$, the equation above has to be intended as $Q_{ij}^{(\beta)} = 0$. Also, $Q_{jj}^{(\beta)}$ is fixed by $Q_{jj}^{(\beta)} = 1 - \sum_{i \neq j} Q_{ij}$.

In this case, what would you expect the stationary distribution $\rho^{(\beta)}$ to be? What happens to the stationary distribution in the limits $\beta \to 0$ and $\beta \to +\infty$?

- Looking at the definition of Q_{ij}^β we recognize the Metropolis rule with proposal kernel equal to the old transition matrix Q_{ij} and the acceptance rate given by $\min\left(1, \frac{k_i^\beta}{k_j^\beta}\frac{Q_{ji}}{Q_{ij}}\right)$. Since the acceptance rule contains the stationary distribution, by construction, the stationary distribution for the generalized PageRank algorithm is given by

$$\rho_i^{(\beta)} = \frac{k_i^\beta}{\sum_{i'} k_{i'}^\beta}$$

where the denominator is just a normalization factor.

For $\beta = 1$ we obtain the standard PageRank. In the limit $\beta \to 0$ we obtain the uniform distribution over the web pages:

$$\rho_i^{(0)} = \frac{1}{N}$$

In the limit $\beta \to +\infty$ the support of ρ concentrates on the webpages with the highest degree. Calling $k^* = \text{argmax}_i\,(k_i)$ the highest degree and $V^* = \{i : k_i = k^*\}$ the set of nodes with highest degree, we have

$$\rho_i^{(\infty)} = \begin{cases} \frac{1}{|V^*|} & \text{if } i \in V^* \\ 0 & \text{otherwise} \end{cases}$$

1. Objective function:

$$f(\hat{x}) = -\frac{1}{b}\sum_{i=0}^{n-1}\log\left[\sum_{j=0}^{k-1}\exp\left(-b\ (\hat{x}_j - x_i)^2\right)\right]$$

where $b > 0$ is some number.

- Given a generic point $\hat{x}$, compute the gradient of f at $\hat{x}$, for each component with index $j \in \{0, 1, \ldots, k-1\}$

$$\text{answer:}\quad (\nabla f(\hat{x}))_j = \sum_{i=0}^{n-1}\left[\frac{2\,(\hat{x}_j - x_i)\exp\left(-b\ (\hat{x}_j - x_i)^2\right)}{\sum_{j'=0}^{k-1}\exp\left(-b\ (\hat{x}_{j'} - x_i)^2\right)}\right]$$

Notice that, since the index j is being used on the left hand side of the expression, we need to use a different index name in the summation in the denominator, and we have just used j'.

$$\begin{aligned}
\frac{\partial}{\partial x_j}\left(5\sum_{i=0}^{n-2}(x_i\ x_{i+1})^2\right) &= \left(5\sum_{i=0}^{n-2}\frac{\partial}{\partial x_j}(x_i\ x_{i+1})^2\right) \\
&= 5\sum_{i=0}^{n-2}2\,(x_i\ x_{i+1})\,\frac{\partial}{\partial x_j}(x_i\ x_{i+1}) \\
&= 10\sum_{i=0}^{n-2}(x_i\ x_{i+1})\left(\frac{\partial x_i}{\partial x_j}x_{i+1} + x_i\frac{\partial x_{i+1}}{\partial x_j}\right) \\
&= 10\sum_{i=0}^{n-2}(x_i\ x_{i+1})\,\frac{\partial x_i}{\partial x_j}x_{i+1} + 10\sum_{i=0}^{n-2}(x_i\ x_{i+1})\,x_i\frac{\partial x_{i+1}}{\partial x_j} \\
&= 10\sum_{i=0}^{n-2}(x_i\ x_{i+1})\,\frac{\partial x_i}{\partial x_j}x_{i+1} + 10\sum_{k=1}^{n-1}(x_{k-1}\ x_k)\,x_{k-1}\frac{\partial x_k}{\partial x_j}
\end{aligned}$$

$$(\nabla f(x))_i = 2x_i\,(x_i - 1)^2 + 2x_i^2\,(x_i - 1) - w_i + 10x_i\begin{cases} x_1^2 & \text{if } i = 0 \\ \left(x_{i+1}^2 + x_{i-1}^2\right) & \text{if } 0 < i < n-1 \\ x_{n-2}^2 & \text{if } i = n-1 \end{cases}$$

The generic formula is:

$$x' = x - \alpha\nabla f(x)$$

In terms of the individual components it's just

$$x_i' = x_i - \alpha\,(\nabla f(x))_i$$

where the expression of $(\nabla f(x))_i$ is written above.

```
def island_of_doom(n, i0, j0, t, trials):
    num_alive = 0
    for trial in range(trials):
        i, j = i0, j0
        alive = True
        for step in range(t):
            direction = np.random.randint(4)
```

```
            if direction == 0: # left
                i -= 1
            elif direction == 1: # right
                i += 1
            elif direction == 2: # up
                j -= 1
            else # direction == 3 # down
                j += 1
            if not (0 <= i < n and 0 <= j < n):
                alive = False
                break
        num_alive += alive
    return num_alive / n
```

Newton method

$$x_1 = x_0 - \alpha \frac{f'(x_0)}{f''(x_0)}$$

```
def mincoins(l, x):
    if x == 0:
        return 0
    best = None
    for c in l:
        if c > x:
            continue
        n = 1 + mincoins(l, x-c)
        if best is None or n < best:
            best = n
    return best
```

Recursion, complexity, proof by contradiction

LOOK FOR “TASK”

[automatic balancing can be implemented, at a cost, but it's not implicit in the definition of a BST] [it's relatively efficient but not O(1), at best it's O(log(n)) if the tree is balanced]

If proposal probabilities are not symmetric, the stationary distribution of the MCMC process at fixed beta is not Boltzmann distribution.