0.1 Fibonacci heaps

- *Algorithm:* Fibonacci heaps
- *Input:*
- *Complexity:* $O(logn)$ for EXTRACT-MIN and DELETE, $O(1)$ for other operations
- *Data structure compatibility:* Fibonacci heap as a data structure, implemented through linked list
- *Common applications:* Dijkstra's shortest path algorithm

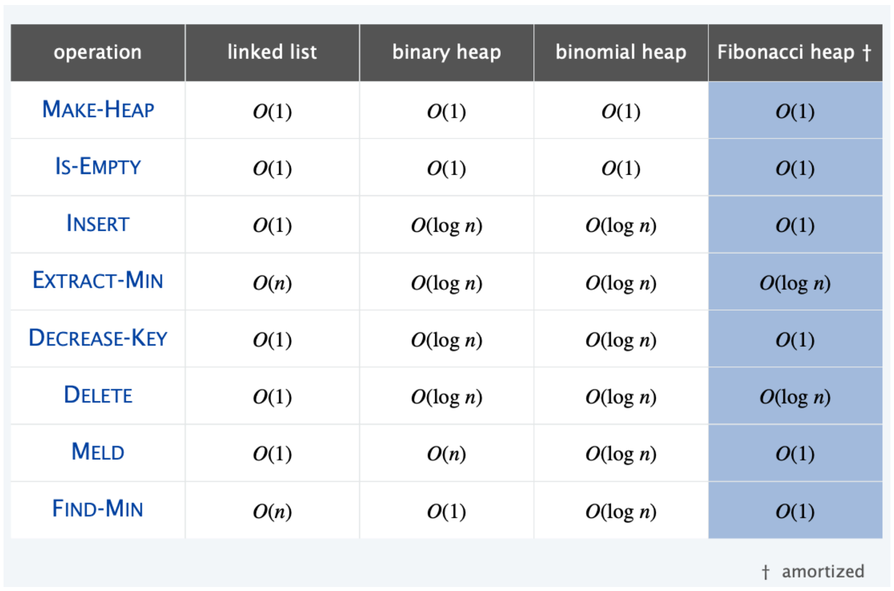
Problem. Fibonacci heaps

The original motivation is to improve the Dijkstra's shortest path algorithm. First, it supports a set of operations that constitutes what is known as a "mergeable heap." Second, several Fibonacci-heap operations run in constant amortized time, which makes this data structure well suited for applications that invoke these operations frequently [**1**].

Description

As a data structure that constructs a mergeable heap, it supports operation *MAKE-HEAP(), INSERT(H,x), MINIMUM(H), EXTRACT-MIN(H), UNION(H_1, H_2), DECREASE-KEY(H,x,k), DELETE(H,x)*, where H is a fibonacci heap and x is an element, k is a new key value.

As a implementation of priority queue, its has good amortized time complexity, compare to linked list, binary heap or binomial heap (*MELD* stands for *UNION*) [1].



operation	linked list	binary heap	binomial heap	Fibonacci heap †
MAKE-HEAP	$O(1)$	$O(1)$	$O(1)$	$O(1)$
IS-EMPTY	$O(1)$	$O(1)$	$O(1)$	$O(1)$
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
EXTRACT-MIN	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECREASE-KEY	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
DELETE	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
MELD	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$
FIND-MIN	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$

† amortized

Figure 1: Comparison of Priority Queue Performance

Structure:

- A set of heap ordered trees, such that in each tree, $child.key \geq parent.key$.
- Heap store a pointer of the minimum node. Root list is maintained through circular, doubly linked-list.
- Node has a pointer to its parent; a pointer to any of its children; a pointer to its left and right siblings; its degree (number of children); whether marked.

Analyze of Time complexity:

For a given Fibonacci heap, potential function: $\Phi(H) = trees(H) + 2 \cdot marks(H)$. For each operation, analyze the actual cost, change in potential and amortized cost.

- *Insert:* $c_i = O(1), \quad \Delta\Phi = \Phi(H_i) - \Phi(H_{i-1}) = +1, \quad \hat{c}_i = c_i + \Delta\Phi = O(1)$
- *Extract Min:* $c_i = O(rank(H)) + O(trees(H)), \quad \Delta\Phi \leq rank(H') + 1 - trees(H), \quad \hat{c}_i = c_i + \Delta\Phi = O(logn)$
- *Decrease Key:* Denote the number of cuts as c, $O(1)$ for each cut and meld it into root list, $c_i = O(c), \quad \Delta\Phi = O(1) - c, \quad \hat{c}_i = c_i + \Delta\Phi = O(1)$
- *Union:* $c_i = O(1), \quad \Delta\Phi = 0, \quad \hat{c}_i = c_i + \Delta\Phi = O(1)$
- *Delete:* Achieved by *Decrease Key* and *Extract Min*, so $\hat{c}_i = O(rank(H))$

How to analyze rank(H)

A Fibonacci heap with n elements, then $rank(H) \leq log_{\Phi} m$, where Φ is the golden ratio $= (1/\sqrt{5})/2 \approx 1.618$.
Discussion:

The difficulty of implementation and the constant factor make it not so practical in real life. It will be useful for large data size. The $\Theta(1)$ amortized time of *Decrease Key, Extract Min, Delete* make it desirable when number of such operations needed. So computing minimum spanning tree, finding single-sourse shortest paths make essential use of it.

It is named as "Fibonacci Heap" because Fibonacci numbers are used in running time analysis. Every node in Fibonacci Heap has at most degree at $O(logn)$. The size of a subtree rooted in a node of degree k is at least F_{k+1}, where F_k is the $k-$th Fibonacci number.

References.

[1] Pearson. *Lecture Slides for Algorithm Design.* URL: http://www.cs.princeton.edu/~wayne/kleinberg-tardos (cit. on p. 1).

0.2 Support Vector Machine

- *Algorithm:* Support Vector Machine (algo. 1)
- *Input:* set of (input, output) training pair samples, where input has n features such that $x_1, x_2, ..., x_n$ and output as y.
- *Complexity:* $\mathcal{O}(n^2)$ for linear, $\mathcal{O}(n^2) - \mathcal{O}(n^3)$ for nonlinear.
- *Data structure compatibility:* N/A
- *Common applications:* Machine Learning, for data classification,

Problem. Support Vector Machine

Given training vectors $x_i \in \mathbb{R}^p$, i=1,. . . , n, in two classes, and a vector $y \in \{1, -1\}^n$, our goal is to find $w \in \mathbb{R}^p$ and $b \in \mathbb{R}$ such that the prediction given by sign$(w^T \phi(x) + b)$ is correct for most samples. .

Description

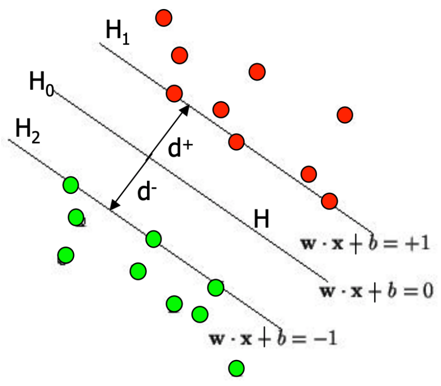
Basic idea: The objective of support vector machine algorithm is to find a hyperplane in an N-dimensional space (N is the number of features). Then this optimal hyperplane could be used for linearly separable patterns (distinctly classifies the data points).

More specific, the hyperplanes is the decision boundaries used to classify the data points. For example, if the number of input features is 2, then a *line* is the hyperplane in a 2-dimensional space, which is binary classification. So with the **input** as a set of (input, output) training pair samples, where input has n features such that $x_1, x_2, ..., x_n$ and output as y, the **output** is a set of weights $\mathbf{w}$ $(w_1, w_2, ..., w_n)$, one for each feature. Then the linear combination of those weights predicts the value of y.

Important Concepts:

- **Support Vector:** Data points that are close to the hyperplane, influence the position and orientation of the hyperplane, which could be used to maximize the margin of the classifier [2]. This is a key difference between SVM and neural nets. Because there are finite sets of weights $\mathbf{w}$ that we could choose from, all the data points will influence the optimality for neural nets, while SVM only consider the support vectors (so other points will not affect the boundary).
- **Hyperplane** H defined as [3]

$$w \cdot x_i + b \geq +1 \text{ when } \mathrm{y_i} = +1$$
$$w \cdot x_i + b \leq -1 \text{ when } \mathrm{y_i} = -1$$
$$H_1 : w \cdot x_i + b = +1$$
$$H_2 : w \cdot x_i + b = -1$$
$$H_0 : w \cdot x_i + b = 0$$



H_1
H_0
H_2
d^+
d^-
H
$\mathbf{w \cdot x} + b = +1$
$\mathbf{w \cdot x} + b = 0$
$\mathbf{w \cdot x} + b = -1$

$d+/d-$: the shortest distance to the closet positive/negative point. And no points between H_1 and H_2. Points on plane H_1 and H_2 are support vectors.

- **Optimal Hyperplane:** H with the maximum margins. Notice the distance between H_0, H_1 is $2/||w||$, so need to minimize $||w||$

Minimize $||w||$ and no points between H_1, H_2 together defined a constrained optimization problem, which could be solved by the Lagrangian multiplier method. Also we could solve it by just computing the inner products of x_i, y_i [3].

We minimize $||w||$ while add penalty when sample misclassified (penalty strength controlled by C). Allow some distance ζ_i between points and correct boundary. $\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta_i$ subject to $y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i, \zeta_i \geq 0, i = 1, ..., n$ [4]. The function finally obtained to optimize is $L_d = \sum a_i - 1/2 \sum a_i a_j y_i y_j (x_i \cdot x_j)$.

Non-linear SVMs: As we compute $(x_i \cdot x_j)$, if transform the vector to get a linear situation, need to compute $\phi(x_i), \phi(x_j)$, which will be time consuming. However, with **kernel function** K such $K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$, no need to know or compute ϕ. The kernel function defines inner products in the transformed space [3]. The function finally obtained to optimize is $L_d = \sum a_i - 1/2 \sum a_i a_j y_i y_j K(x_i, x_j)$. Generally speaking, for a nonlinear SVM is between $\mathcal{O}(n^2)$ (small C) and $\mathcal{O}(n^3)$ (large C) with n amounts of training instances [1].

Complexity:

The core of an SVM is the separation between support vectors and the rest of the data. The training time complexity depends on the number of samples, type of the kernel function and the penalization C (regularization parameter). For linear case, generally could expect training time of $\mathcal{O}(n^2)$. It is highly depends on how we solve the quadratic problem and choose the support vectors. The training time for a linear SVM to reach a certain level of generalization error actually decreases as training set size increases

To demonstrate, consider a simple two dimensional case

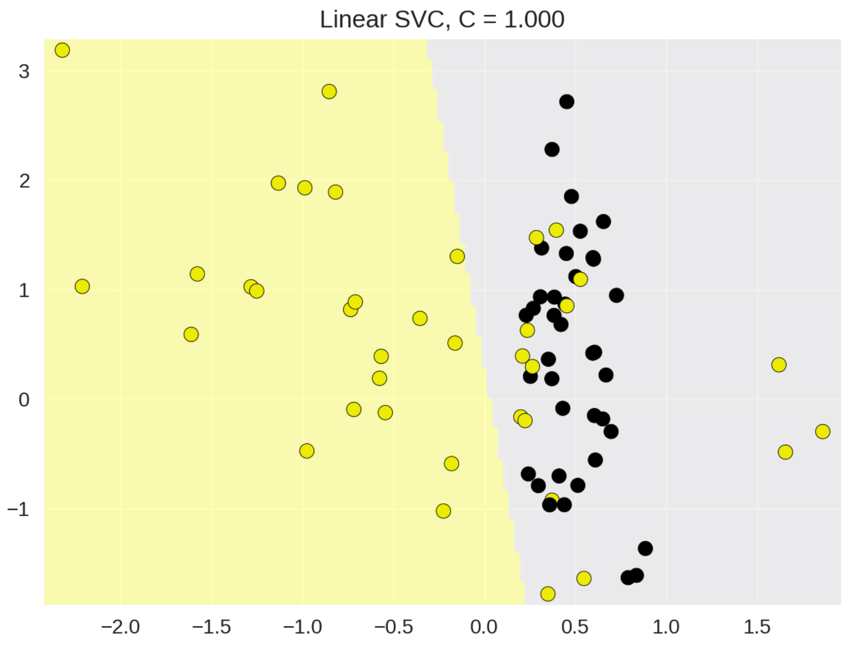
This simple binary classification problem has two informative features (data points generated by *Python* $make_classification$). So the two classes are separable by a linear classifier. yellow and black points represent

Algorithm 1: Training an SVM

Input : $\mathbf{X}$ and y loaded with training labeled data, $\alpha \leftarrow 0$ or partially trained SVM
Output: trained SVM

```
1 $C \leftarrow$ value specified by the case
2 while changes in $\alpha$ and other resource constraint not met do
3     for $\forall\ (x_i, y_i), (x_j, y_j)$ do
4         Optimize $\alpha_i$ and $\alpha_j$
5     end for
6 end while
```

different class of points. It illustrates how C parameter performs regularization. Large C represent less regularization and will fit the training set with as few errors as possible, even if it means using a small immersion decision boundary. Very small values of C on the other hand use more regularization that encourages the classifier to find a large marge on decision boundary, even if that decision boundary leads to more points being misclassified.



References.

[1] L´eon Bottou. *Support Vector Machine Solvers.* URL: https://leon.bottou.org/publications/pdf/lin-2006.pdf (cit. on p. 3).

[2] Rohith Gandhi. *Support Vector Machine — Introduction to Machine Learning Algorithms.* 2018. URL: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47 (cit. on p. 3).

[4] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on p. 3).

[3] Village Idiot R. Berwick. *Guide to Support vector Machines (SVMs).* 2019. URL: http://web.mit.edu/6.034/wwwbob/svm-notes-long-08.pdf (cit. on p. 3).

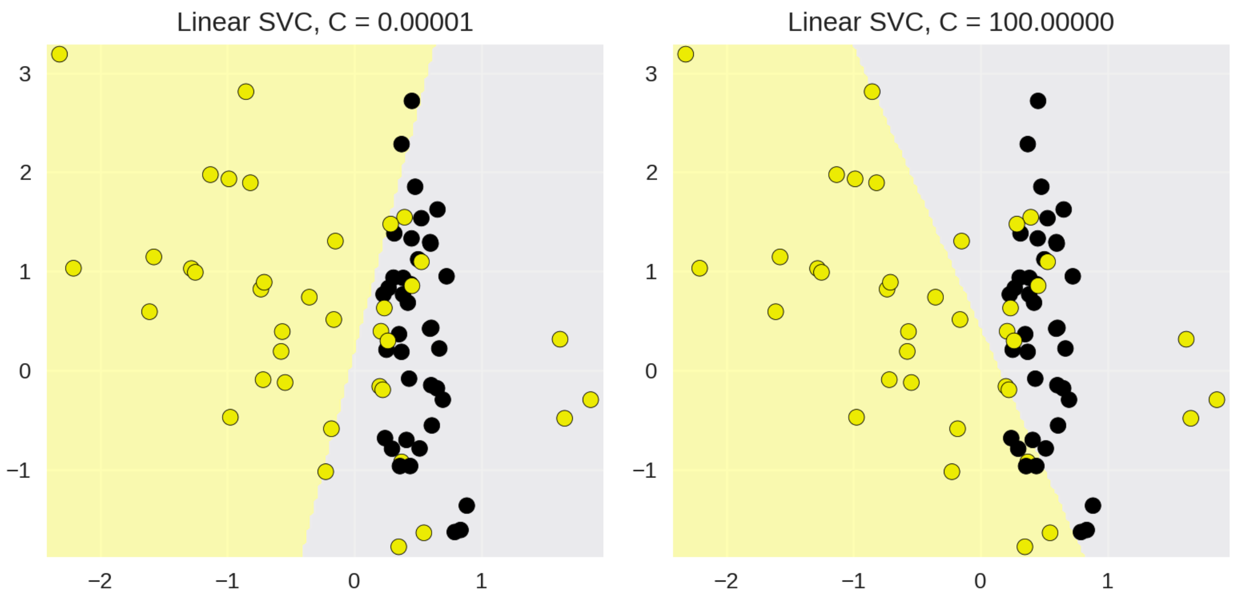


Figure 2: C parameter affect SVM