MATH 441 Learning Portfolio

Alex Golab

December 14, 2022

1 Convexity Proof

The proof of convexity is a very important proof in linear programming, convex optimization and machine learning. A set is said to be convex if it satisfies the following properties:

A set $S \subset \mathbb{R}^n$ is convex
if $\forall \vec{x}$, $\vec{y} \in S$, $t \in [0,1]$, $(1-t)\vec{x} + t\vec{y}$ is also in S

The formula is really just a complicated way of saying that a set is convex if the line between any two points in the set is still in the set. For example, the set on the left would be convex since no line between two points in the set travels outside the set, while the set on the right violates this condition and therefore isn't convex.

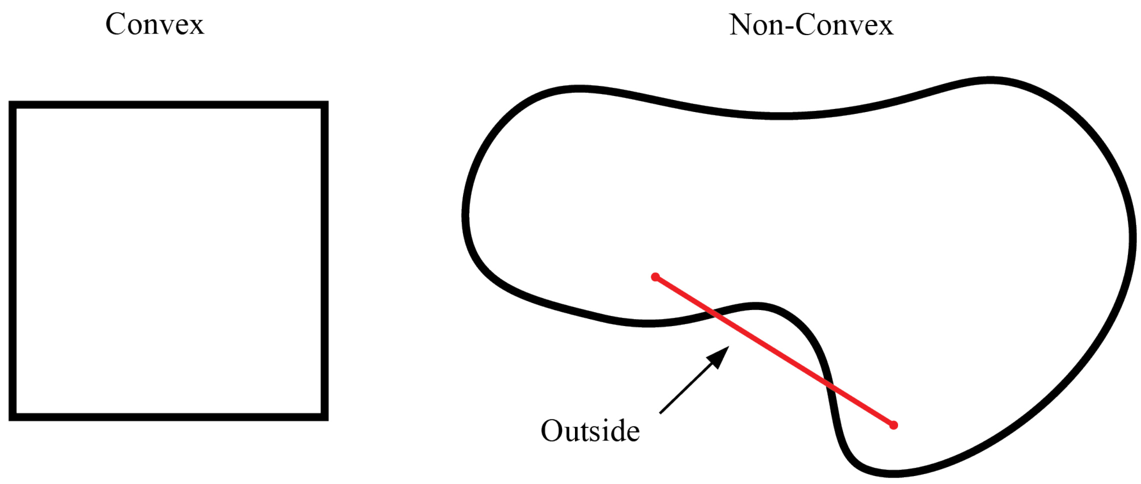


Figure 1: Convex vs Non-convex

A good example of applying this proof is on a half circle. we can define a half circle as the following: $H = \{\vec{x} \in \mathbb{R}^n | \vec{a} \cdot \vec{x} \leq b\}$

Proof. Let $\vec{x}, \vec{y} \in H$. Since $x, y \in H$ by the definition of a half circle $\vec{a} \cdot \vec{x} \leq b$ and $\vec{a} \cdot \vec{y} \leq b$. If the line $(1-t)\vec{x} + t\vec{y}$, $t \in [0,1]$, is $\in H$, then $\vec{a} \cdot (1-t)\vec{x} + t\vec{y} \leq b$. This can be shown as follows:

$$\begin{aligned}
\vec{a} \cdot (1-t)\vec{x} + t\vec{y} &\leq b && (1)\\
= (1-t)\vec{a} \cdot \vec{x} + t\vec{a} \cdot \vec{y} &\leq b && (2)\\
\leq (1-t)b + tb &\leq b && (3)\\
= (1-t+t)b &\leq b && (4)\\
= b &\leq b && (5)
\end{aligned}$$

Therefore the half circle H is convex □

We can then apply this proof to linear programming problems. Since the domain of linear programming problems always follow the form $Ax \leq b$ we get the following proof:

Proof. let $\vec{x}, \vec{y} \in$ a linear programming problem. Therefore $A\vec{x} \leq b$ and $A\vec{y} \leq b$. If the line $(1-t)\vec{x}+t\vec{y}$, $t \in [0,1]$, is $\in$ our linear problem then $A((1-t)\vec{x}+t\vec{y}) \leq b$. This can be shown as follows:

$$
\begin{aligned}
A((1-t)\vec{x}+t\vec{y}) &\leq b && (6)\\
= (1-t)A\vec{x}+tA\vec{y} &\leq b && (7)\\
\leq (1-t)b+tb &\leq b && (8)\\
\leq (1-t+t)b &\leq b && (9)\\
= b &\leq b && (10)
\end{aligned}
$$

Therefore linear programming problems are convex. □

Proving convexity for linear programming problems (and convex optimization and machine learning problems) is particularly important since we are looking for extrema. In non-convex problems, we can not know if the extremum we are currently at is global or local, and since we cannot know the value of all other extrema unless we calculate their values we do not know if we have truly found an optimal result. As an example, this would be like standing at the bottom of a valley surrounded by mountains. It will look like we are at the lowest point since the ground looks to raise in every direction, but we do not truly know. The lowest point of the mountain could just as easily lie at the bottom of any of the other valleys surrounding it, and we would have to check everyone to find out (something not feasible to do in many cases).

[4]

2 Comparing Search Algorithms

2.1 Basic Search

In class we went over a type of search called branch and bound, I know from other classes that more search algorithms exist and I wanted to take this chance to explore them. Search algorithms follow a basic structure, which we can then mutate into our various algorithms.

Algorithm 1 My algorithm

```
frontier ← [$< n_0 >$]
while frontier not empty do
    select and remove path $< n_0, \ldots, n_k >$ from frontier
    if goal == $n_k$ then
        return $< n_0, \ldots, n_k >$
    else
        for all neighbours $n$ of $n_k$ do
            add $< n_0, \ldots, n_k, n >$ to frontier
        end for
    end if
end while
return null
```

At a glance, this search algorithm seems to be very basic, however, due to the freedom we are left with when choosing what order items enter and leave the frontier we can drastically change the way we search through a graph.

Before going further, it is worthwhile to explain what a "frontier" is. Simply put, the frontier is a list of paths. The last node in a path on the frontier is the current node we are searching with all nodes coming before being nodes we've explored, and everything after being nodes we haven't looked at yet.

This idea is illustrated in the image below.

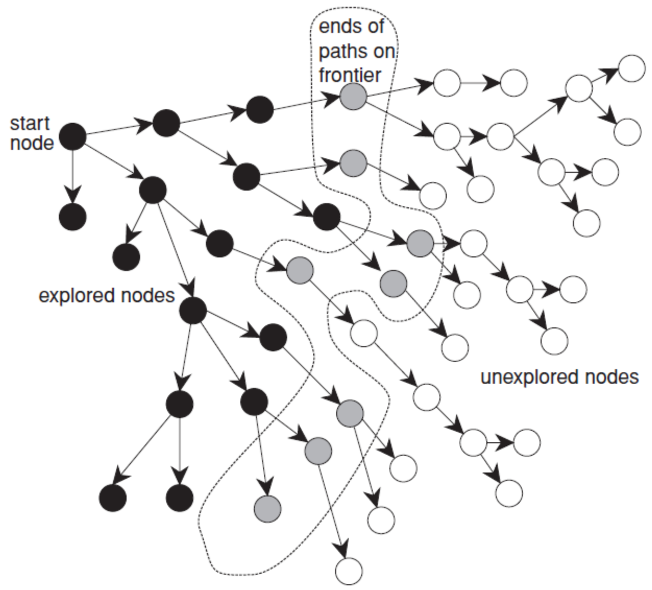
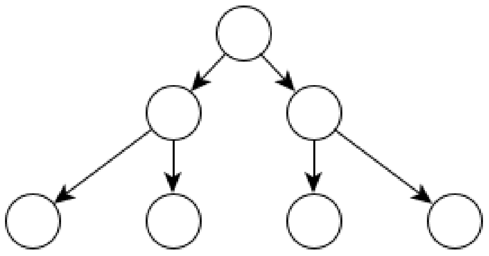


Figure 2: https://www.skedsoft.com/books/artificial-intelligence/a-generic-searching-algorithm

In order to analyze search algorithms is to establish how we measure performance. It is easiest for us to compare how long algorithms take by using trees. To describe these trees we define d as the depth of the tree (the number of edges traversed) and b the branching factor. The tree included below would have a depth of 2 and a branching factor of 2.



We will also be using big O to measure run times with $f(x) \in O(g(x))$ if $\exists M > 0$, x_0. s.t. $f(x) \leq M * g(x)$ for $x \geq x_0$. In simpler terms, we are always saying that $f(x)$ is smaller than $g(x)$ or that it's an upper bound.

2.2 BFS

BFS or breadth-first search is the result of treating the frontier as a queue (the first item on is the first item off), with the first path on being the first path we take off to explore. This results in us taking one step and then switching to another path causing us to slowly progress over a large selection of paths. A gif has been included as a visual example below (only some PDF views can play gifs. The caption is a link to watch if the gif won't play).

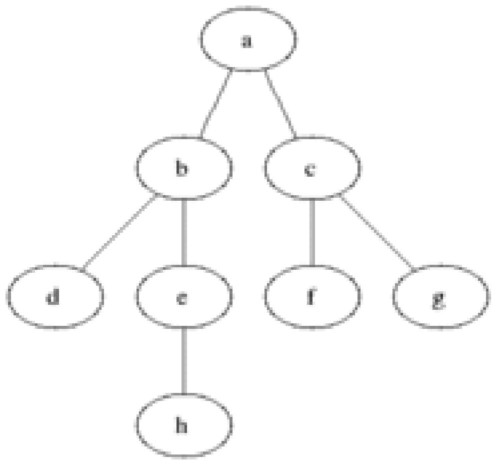


Figure 3: `https://upload.wikimedia.org/wikipedia/commons/4/46/Animated`$_BFS.gif$

Since BFS looks at the nodes in a graph at increasing depth, it will always return the solution with the shortest path (so long as it's unweighted).

In the worst case, BFS will have to look at all the nodes in the graph, which means we have a time complexity of $O(b^d)$. In the case of space complexity, in the worst case if BFS is just about to start exploring the nodes on the bottom row, paths from the start to each one of the bottom nodes will be on the frontier, meaning we will have to store the entire graph on the frontier giving us a worst-case space complexity of $O(b^d)$.

2.3 Depth First Search

Depth First Search or DFS is the second of our basic unweighted search algorithms. Unlike BFS the frontier in DFS acts as a stack in which the last item onto a stack is the first off. The use of a stack instead of a queue is where we get the "depth" in the depth-first search, as DFS will take a node and explore it until completion before following another path.

Just like BFS in the worst case, DFS will have to look at every node, which will again be $O(b^d)$. However, unlike BFS since DFS will follow one and only one path to completion (in other words its max depth) before starting a new path, it only needs to store the path it's looking at (d nodes) and the next steps in the paths it needs to look at, or the other branches (b per node). Thus we get a space complexity of $O(b * d)$

Unfortunately, DFS unlike BFS won't always return the best solution. In the following image, it would return the deeper solution on the earlier path than the shallower one on the latter path. Another problem of DFS is that it can get stuck going in circles since, unlike BFS which rotates between multiple paths, DFS only focuses on one path. We can get around this by implementing cycle checking, but that gets rid of the only advantage that DFS had, which is its space complexity. We can combine the two, into what we call Iterative Deepening.

2.4 Iterative Deepening

Iterative deepening (IDS) is the exact same as DFS, except we have a limit on how deep we can go before we have to look at a different path. If no solution is found we restart with a deeper maximum depth. This algorithm takes advantage of BFS (its optimal) and combines it with DFS (its space complexity).

A simple proof is as follows:

Proof. Let X be an instance of IDS which returned solution s_1 at depth k for graph G. Let s_2 be another solution on the graph with depth $d, d < k$. This is a contradiction since IDS if IDS returned a solution of depth k it means it looked at all nodes with depths $1, 2, 3, \ldots, k-1$ and would have returned a solution, meaning either s_2 isn't a valid solution or $d \geq k$. □

Another surprising fact about IDS is that it has the same time complexity as DFS and BFS, which you would think isn't the case since every time the depth increases the search restarts. The contrary to this assumption can be shown as follows.

The table shows the depth of the nodes, the number of different possible paths at this node, and the number of times we will have to look at this node once we reach a depth of d

Depth	# of paths	# of times at depth d	Total number of evaluations at depth
1	b	d	db
2	b^2	$d-1$	$(d-1)b^2$
3	b^3	$d-2$	$(d-2)b^3$
.	.	.	.
.	.	.	.
.	.	.	.
d	b^d	1	b^d

If we sum the total number of paths evaluated at each depth (the last column) we get the total number of paths evaluated total or the run time.

$$b^d + 2b^{d-1} + 3b^{d-2} + \cdots + db = b^d(1 + 2b^{-1} + 32b^{-1} + \cdots + db^{1-d})$$
$$= b^d \sum_{i=1}^{\inf} ib^{1-i} = b^d \frac{b}{b-1}^2 = O(b^d)$$

Interestingly enough is the same time complexity as BFS and DFS. This means that we've essentially fused both BFS and DFS keeping advantages from both. This doesn't mean that it's always best to use IDS, for example, if we have solutions sitting at a known depth or if we know the solution is deep IDS would still look at all shallower nodes before the solution. Despite this in most cases for uninformed search, it is a strong algorithm.

This was just a short overview of simpler search algorithms. These search algorithms only work for graphs without costs, as searching on graphs of that type involves going into heuristics and other concepts.

[3]

3 3-Partition Problem

The 3-Partition problem is a subset of set partition problems. The problem states that given a set of integers, is there a partition of triplets such that all sum to the same value.

For this problem we have:

- Input: a set S of $n = 3m$ positive integers. All numbers should sum to mT if we want this to be $\mathcal{NP}$-Hard
- Output: A true or false and m subsets

Code is in an attached Jupyter Notebook

To solve this problem I decided to generate the problem space as a graph with each layer being an assignment of each number in the problem list and its position on that layer which subset it goes to. for example the list a, b, c, d, e, f which has 2 subsets would be (partly) graphed as the following:

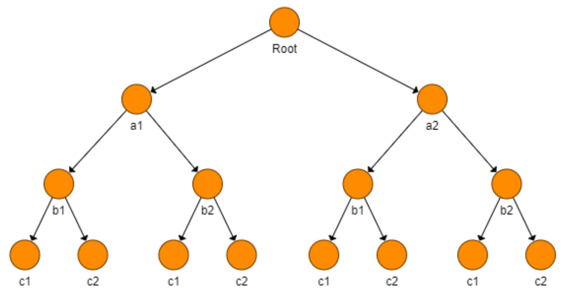


Figure 4: Search Graph for the 3-partition problem

The search we will employ is called DFS with pruning. DFS with pruning was covered earlier in the learning portfolio, but the pruning aspect was skipped. In DFS with pruning, we navigate through the graph via DFS, however, we have a scoring function which checks the solution. In the case where an invalid solution is produced, we prune that path from the search and no longer explore it for a solution. This will allow us to get rid of all paths that have split incorrectly between subsets or have a subset which sums to an invalid amount without having to waste time exploring the combination further.

Since we are navigating the graph using DFS we gain its time complexity of $O(b^d)$ possible nodes we will have to explore in the worst case. This isn't entirely amazing of a run-time as it sounds as solving the problem also requires forming a search tree.

[5]

4 Textbook Review - Algorithm Design by Jon Kleinberg and Eva Tardos

Algorithm Design by John Kleinberg and Eva Tardos is unsurprisingly a textbook about computer science algorithms. The textbook is aimed at computer science students, however, the topics covered in the textbook also apply to our math class.

The book is designed for the senior undergraduate level over a one or two-semester course and assumes some basic background knowledge of elementary data structures, though plenty of information is accessible from the textbook without this. The textbook follows a standard accumulative module pattern, with each chapter building on previous ideas. Each chapter introduces a problem to be solved and walks the reader through, how to interpret the problem, how to represent the problem, and finally how to solve it. The book further goes into detail analyzing the runtime and the complexity of the algorithm, opening the door to further optimization or alternative approaches (usually explored in later chapters).

The goal of this book is to give the reader a basic intuition to solve problems through the creation and optimization of computer algorithms. The textbook starts by establishing practical problems to explore throughout the book, giving the readers a basic intuition on how to approach the problems. The book then proceeds to define how we measure the efficiency of an algorithm, giving us the tools to compare performance and measure if we have actually created a successful algorithm. The final introductory module the book goes through is defining graphs and how we represent and work with them, before finally stepping into our first algorithms.

Algorithm Design covers: greedy algorithms, clustering, divide and conquer algorithms, dynamic programming, network flow, P and NP, P-Space, approximation algorithms, local search, and finally randomized algorithms. Throughout the book, the 5 basic problems described in the beginning are explored through the various topics (when compatible), as well as novel problems added in each module. By applying these different approaches to solving each we can compare and contrast the advantages and disadvantages of each.

In general, I think that this book provides a well-paced introduction to algorithm design and is a great introduction to the field. In the case of math 441, I think the sections on greedy algorithms, network flow, P and NP, and approximation algorithms are particularly applicable. The textbook does demand that the reader already has a grasp on basic set and graph theory, alongside

5 $\mathcal{P}$ and $\mathcal{NP}$ Reductions

During the lesson on P and NP, one of the interesting parts of the talk was that we could collapse all $\mathcal{NP}$ problems to $\mathcal{P}$. I remember a bit about this from "reductions" taught in a previous class but wanted to explore this further here.

Before exploring $\mathcal{P}$ and $\mathcal{NP}$ we should clarify what I mean by reduction.

5.1 Reductions

Reductions are a simple yet powerful tool. A reduction of problem X to problem Y, commonly denoted as $X \leq_P Y$ is when you take the inputs into problem X and convert them into inputs to problem Y in polynomial time. By reductions, if we have a problem X which we don't know how to solve in polynomial time and we have a problem Y which we can solve in polynomial time, if we can reduce X to Y in polynomial time and convent back in polynomial time we can by proxy solve X in polynomial time.

5.2 $\mathcal{P}$ Problems

If a problem X is considered to belong to $\mathcal{P}$ ($X \in \mathcal{P}$) means that we have some algorithm A that can solve X in polynomial time.

An example of a problem that belongs to $\mathcal{P}$ is the String Matching problem. In this problem, you take two strings a and b and see if a is a contiguous subset of b.

For example, take the string **base** and **The baseball game is starting**. Solving this problem is easy all we have to do is check string a at each position in b and see if it matches.

0: **The baseball game is starting**
1: **base**
2:   **base**
3:     **base**
4:       **base**
5:         **base *Success***

If a has a length of u characters b has a length of v, we can try at most $v - u + 1$ positions. since we only have to check the length of string a each check only takes u steps. From this, we get a total time of $u(v - u + 1)$. Using a bit of math we know that the steps are longest when $v \approx 2u$, meaning that $un/3$ (since $n = u + v$, the total number of characters). This means that $u(v - u + 1)$ becomes $u(u + 1) \approx (n/3)(n/3)$ which is $O(n^2)$. Because we now have an algorithm A that solves X in polynomial time we know X belongs to $\mathcal{P}$.

5.3 $\mathcal{NP}$ Problems

A problem X is considered to be in $\mathcal{NP}$ ($X \in \mathcal{NP}$) if there exists an efficient certifier C (something that can check whether a solution is correct in polynomial time). To do this C needs a polynomial function p that can check whether two inputs, both an input to X, labelled s, and

For example the **Hamiltonian Cycle Problem**. In this problem, given a graph, you are required to find a path to every node without revisiting a node (minus the start), or in other words to visit each node once.

Solving this problem is hard, but verifying it isn't, which is the intuition for a problem belonging to $\mathcal{NP}$. In this case, if the graph we are constructing a Hamiltonian cycle on has n nodes to verify that the presented solution is valid we just have to check each node once and verify that every node has only two edges. This means we only have to look at all n nodes once, which is $O(n)$ time or linear. Since linear is an efficient certifier it means that the Hamiltonian Cycle problem belongs to $\mathcal{NP}$.

5.4 $\mathcal{NP}$-Hard and $\mathcal{NP}$-Complete Problems

We call a problem $\mathcal{NP}$-Hard if all problems in $\mathcal{NP}$ reduce to the problem. An $\mathcal{NP}$-Complete problem, is just any that belongs to $\mathcal{NP}$ and to which every problem in $\mathcal{NP}$ can be reduced. If a problem does not belong to $\mathcal{NP}$, but every problem in $\mathcal{NP}$ can be reduced to it then we call this problem $\mathcal{NP}$-Hard.

Now it's easy to get carried away and to start trying to prove that our problem is $\mathcal{NP}$-hard, but it's worthwhile to think about whether or not it's even possible. It's not hard to imagine that there could be two algorithms incomparable to each other exist that could never reduce to the same algorithm. Luckily, however, once we find one, we know that the rest must also exist, and luckily someone has already solved it.

The first proof was done with Circuit Satisfiability, which when given a boolean (true or false) circuit involves finding an assignment to the inputs that results in a true output. An abridged version of the proof is as follows:

Proof. Consider $X \in \mathcal{NP}$. The intuition behind $X \leq_P$ Circuit Satisfiability is that any problem that can be represented as n bits outputs a yes or no can be reduced to a Circuit Satisfiability problem. If a problem can be implemented on a physical computer, then it will be represented as n bits when it reaches the processor of the computer with an output of true or false output. Putting this together means that any problem that can be represented on a physical computer can be reduced to Circuit Satisfiability.

Now to complete this we need an efficient certifier C that checks whether the solution is correct in polynomial time. To make this certifier we would input both the proposed solution as n bits and our certifier string of $p(n)$ bits into a black box of circuit satisfiability. This will be some input of $n + p(n)$ bits which are polynomial and to check whether the solution is valid would be trivial since we just have to look at the output boolean and whether it's true (a valid solution) or false.

$\square$

5.5 $\mathcal{NP} = \mathcal{P}$

Now that we know what NP-completeness how to apply this to prove that $\mathcal{P} = \mathcal{NP}$ would be as follows.

Proof. Let X be an $\mathcal{NP}$-complete problem. If we have an algorithm that can solve X in polynomial time that means that $\mathcal{P} = \mathcal{NP}$. Furthermore, let algorithm Y be any other algorithm that belongs to $\mathcal{NP}$. Since X is $\mathcal{NP}$-complete $Y \leq_P X$ meaning Y can be solved in polynomial time. Since this is for all other algorithms in $\mathcal{NP}$ is shows that $\mathcal{P} = \mathcal{NP}$ $\square$

The way to interpret this is that if we can reduce any problem in $\mathcal{NP}$ to a $\mathcal{NP}$-complete problem, then all we need to do is find one $\mathcal{NP}$-complete problem that we can solve in polynomial time and we can reduce all other problems in $\mathcal{NP}$ to said problem, solving all of them in polynomial time.

6 L^1 Norm

Included as a Jupyter notebook

[1] [2]

7 Intro to Compressed Sensing

A video that can be viewed at https://youtu.be/0eNgC-EYy3c

[2]

References

[1] P. Bauch. Linear regression demo.

[2] S. L. Brunton and J. N. Kutz. Data-driven science and engineering: machine learning, dynamical systems, and control.

[3] D. Poole and A. Mackworth. Artificial intelligence: Foundations of computational agents 2e. 2017.

[4] R. J. Vanderbei. Linear programming: Foundations and extensions 5e. 2020.

[5] Wikipedia. 3-partition-problem.