ABSTRACT

So, here’s the thing. Lately, I got pretty interested in web development, but the problem was I have never done it before. Having said that, I am not new to programming. I have worked with languages like C/C++, Java, JavaScript, and my favorite one Python. Python reserves a special place in my heart.
Okay, let’s get to web development and the project I made and the experience I got making it.
My interest in web development grew from the hype and enthusiasm shared by the Vue community. So, I did the basic HTML and CSS so that I can jump right into the land of Vue. And, I have to say that the praise Vue has got is worth it. It is so easy to delve right into it from the moment you learn it. Thanks to the simplicity built into the language. For me, this is the same aspect because of which Python is so dear to me.
Okay, enough of talking. Let me show you what I have built.
The project is called a Pathfinding Visualizer, aptly because it does what it says, it finds a path from a source to a destination. This project is based on graph theory.
Note: I was inspired to do this project after I went to the website built by Clement Mihailescu. At first, I built a version of this using the Processing framework, but later I learnt web development and built the web version.
Algorithm visualization has been high topic in Computer science education for years, but it did not make its way to schools/collages lecture halls as the main educational tool. The present paper identifies two key circumstances that an algorithm visualization must fulfil to be successful: general availability of used software, and visualization of why an algorithm solves the problem rather than what it is doing. One possible method of “why” algorithm visualization is using algorithm unvarying rather than showing the data conversion only. Invariants are known in Program faultless.
Keywords : Breadth First Search, Depth First Search, Dijkstra's algorithm , A* algorithms
Breadth First Search
It is a graph algorithm which does the traversal in a level order fashion. To put it in simple terms, it goes towards the destination layer by layer. It’s like an onion with many layers. You peel out layer by layer to get to the core of the onion. This algorithm gives the shortest path possible.
Depth First Search
It is another graph traversal algorithm. It uses recursion. This algorithm has preferences for going in a particular direction. To put it simply, let’s say it will go always up until it can’t go further in which case let’s say it will go left and likewise. So is the name depth-first. It’s like taking a piece of big wood. Let’s say I have a wood piece of length 8. At first, I cut it into two halves of 4 each. I take the first half and again cut in into two halves of 2 each. Now I will grab one of my piece with length 2 and cut it into two 1. See how I am always going with the new piece every time. After I get to length 1, I cannot divide it further. So, I will go with the other piece of length 2. After I divide this into two one, I will go with the first piece with length 4 and do the same steps. This algorithm generally does not give the shortest path.
Dijkstra's algorithm
A common example of a graph-based pathfinding algorithm is Dijkstra's algorithm. This algorithm begins with a start node and an "open set" of candidate nodes. At each step, the node in the open set with the lowest distance from the start is

examined. The node is marked "closed", and all nodes adjacent to it are added to the open set if they have not already been examined. This process repeats until a path to the destination has been found. Since the lowest distance nodes are examined first, the first time the destination is found, the path to it will be the shortest path.
Dijkstra's algorithm fails if there is a negative edge weight. In the hypothetical situation where Nodes A, B, and C form a connected undirected graph with edges AB = 3, AC = 4, and BC = −2, the optimal path from A to C costs 1, and the optimal path from A to B costs 2. Dijkstra's Algorithm starting from A will first examine B, as that is the closest. It will assign a cost of 3 to it, and mark it closed, meaning that its cost will never be reevaluated. Therefore, Dijkstra's cannot evaluate negative edge weights. However, since for many practical purposes there will never be a negative edgeweight, Dijkstra's algorithm is largely suitable for the purpose of pathfinding.
A* algorithm
A* is a variant of Dijkstra's algorithm commonly used in games. A* assigns a weight to each open node equal to the weight of the edge to that node plus the approximate distance between that node and the finish. This approximate distance is found by the heuristic, and represents a minimum possible distance between that node and the end. This allows it to eliminate longer paths once an initial path is found. If there is a path of length x between the start and finish, and the minimum distance between a node and the finish is greater than x, that node need not be examined.
A* uses this heuristic to improve on the behavior relative to Dijkstra's algorithm. When the heuristic evaluates to zero, A* is equivalent to Dijkstra's algorithm. As the heuristic estimate increases and gets closer to the true distance, A* continues to find optimal paths, but runs faster (by virtue of examining fewer nodes). When the value of the heuristic is exactly the true distance, A* examines the fewest nodes. (However, it is generally impractical to write a heuristic function that always computes the true distance, as the same comparison result can often be reached using simpler calculations – for example, using Manhattan distance over Euclidean distance in two-dimensional space.) As the value of the heuristic increases, A* examines fewer nodes but no longer guarantees an optimal path. In many applications (such as video games) this is acceptable and even desirable, in order to keep the algorithm running quickly.
EXISTING PROBLEMS
In today’s time there are multiple routes of travelling to a particular destination but we don’t know the most time sufficient route. This problem has still not been solved which can cause a lot of time management crisis and can also lead to wastage of fuels which hence makes it very cost consuming. Since not everywhere there is internet connectivity so once we go offline, we lose our destination path which can create a lot of obvious problems.
Ever since the emergence of the first primitive search algorithms, there has been a solution to the most basic definition of the pathfinding problem, in which there are no spatial or temporal constraints on the solution. Given enough time and memory, even the most basic of brute-force search algorithms can exhaustively search the solution space to find the optimal path between two points.
This definition of the problem is, however, very optimistic. Computers are, by the laws of physics, no Turing Machines, and there is, and will always be, an upper bound on how much memory they can facilitate; thus all pathfinding solutions have spatial constraints. The temporal constraint comes from the simple fact that there is often a querying system that has requested the path for one reason, or another, and

can’t wait indefinitely for an answer.
There are three groups of constraints of interest to this research:
• Temporal: the time it takes a pathfinding solution to find the most optimal path it can generate.
• Spatial: the amount of memory that must be allocated by the pathfinding solution during runtime.
• Path Optimality: the degree to which the resulting path corresponds to the shortest path possible in a Newtonian universe. On a Nintendo DS, the spatial constraints can be considered very hard. All pathfinding algorithms need some form of representation of the world they are supposed to find paths in; a simplified abstraction of the world, which the algorithms can understand. The memory architecture of the DS2 poses great limitations on how much information a world abstraction can contain. Memory available for dynamic allocation during runtime is, for the same reasons, very restricted.
In any video game, responsiveness is of great importance. The speed of the pathfinding algorithm used in an RTS game can greatly affect how responsive the game feels to the user; when a unit is ordered to move, it should, ideally, do so without any noticeable delay. To further add to the temporal constraints, pathfinding is seldom the only concurrent task of a video game, and is often given a small percentage of the processing time, in favor of graphics- and sound processing, and other systems of the game. Considering this, and with several units requesting path information simultaneously being a common thing in most RTS games, the temporal constraints of pathfinding in an RTS game can be viewed as hard.
The path optimality constraint of an RTS game can be considered to be moderate. If a path exists between two points, the pathfinder must generate a result, but not necessarily the shortest possible. As long as the movements of units aren’t prone to peculiarities, or can be viewed as moronic by the player, close enough is often accepted as good enough. When combining the constraints of an RTS game with those generated by the DS, the spatial- and path optimality constraints are transferred largely unchanged. The temporality constraint must, however, be modified, as the limitations of the DS influence the performance of the algorithm. By itself, the DS doesn’t generate any temporal constraints, as there are no requirements on how long time the system can work before it must generate an answer. But, as previously mentioned, much attention must be given the responsiveness of an RTS, and the limitations of the DS make the need for a fast algorithm somewhat more vital than on a modern PC3; making the temporal constraints very hard.
PROPOSED SOLUTIONS
Our visualizer will help solve all the above-mentioned problems. It will give the user most time and cost sufficient, i.e., the shortest route to reach their desired destination. Even if the networks fail the user can still access the map with the help of our offline feature which supports the satellite view as well as the road view in order to reach his/her desired destination. The offline feature would come in very handy for elderly people who sometimes may face issues while using a smartphone with internet connectivity.
In sorting algorithms, the user has to give how many inputs and the set of data for inputs or he/she can generate random array inputs using the Generate Button. Then select the particular algorithm from the list and then the visualization of the selected algorithm is shown with the given inputs. Also, the web-based platform has the feature to visualize the algorithm as per the user’s need.
The proposed system involves the simulation of the different type of algorithms

codes. As you can see, there are no major components besides the three coding languages. Most websites have tools or scripts that require a server on the back-end (like PHP), but it is not necessary in this case since React JS runs right in the user's browser.

HTML5 and CSS are used for the interface. The HTML5 communicates with the React JS code and vice versa to launch the appropriate algorithms and update the interface accordingly, as seen with a single, bidirectional arrow. As the React JS was modified from a functional programming focus to a more object-oriented one, the parts of the HTML5 that did change were the function calls for each button. All of the back-end interaction is abstracted to the various buttons for selecting algorithms and running the animation.

TOOLS AND TECHNOLOGY USED

An efficient algorithm is one that calculates the shortest path with the fewest number of node visitations. Among all the Pathfinding Algorithms, Dijkstra's Algorithms is least efficient as it has no method of cutting down on search space and it made far more node visitations during each path calculation phase. Depending, on the situation A* and D* are the most efficient pathfinding

Algorithms

We are going to build this website with the help of HTML(frontend) and Javascript (backend).

ALGORITHM USED:

Sorting Visualization(bubble sort, insertion sort, Quick sort)

Path Finding

Maze breadth

RESULTS AND OUTPUT

The goal of the sorting visualizer tool is to make use of bars and their comparisons to help understand sorting algorithms. This is user flow which shows how sorting visualizer tools can be used.

CONCLUSION AND FUTURE SCOPE

Thus, with our initiative we aim to create to a software which will save time and money and help the user in reaching their destination as soon as possible.

In future we hope to create an offline app which will have pre-saved routes to popular destinations.

With the shift of remote and digital literacy, a combined platform serving the effective literacy requirements of students is required. Algorithm Visualizer is a combined platform that's a comprehensive result for educators and students to educate and learn online effectively. It substantially focuses on "algorithm visualization", which allows a better understanding of its inflow and operation. The developed platform offers a complete perspective of visualizations for sorting and pathfinding algorithms so far.

Also, the web-based platform can run on small devices while providing the feature to visualize at one's own pace.

As a future scope, more Algorithms for Trees, Graphs, Linked List and many more can be added. To empower learning of the learner concept of community learning through forums, discussion and user-based feedback can be added.

The objective of the platform is to reduce the fear level in the minds of learners especially students regarding Algorithms and Data Structures.

Project Setup

For this tutorial I will be using Vite, which is a tool that helps you start projects way faster than npm create-react-app.

First, install Vite if you haven't done that yet. Create a folder in a well-known directory on your machine. Then using the terminal, navigate to this directory and run these commands:

```
$ cd my-app
$ npm install
```

Now inside the src folder, create a components folder, a contexts folder, and a utils folder – and you're done. Here's a screenshot of what your project should look like:

Add Bootstrap

Now we need to add bootstrap to our project for the buttons and icons we're going to add, since we need to focus on the JavaScript part.

Represent the Cells

For the representation of the grid and the cells, I will be building a two dimensional array of objects containing all the properties we need to represent a certain cell.

Now in the utils folder, create a startingGrid.js file. Here, you'll write a function that returns a grid which is a two dimensional array of objects representing the cells.

Each object will have these properties: x, y, isstarting, istarget, iswall, and weight. x and y represent the coordinates of the cell. isstarting is a Boolean that's only true for the starting cell. istarget is similar, but for the target node. Is wall is a Boolean that's only true for walls. And weight is an integer.

All normal cells have a weight of 1, and the weighted cells have a weight of 5. The function should look like this:

```
exportfunction getGrid(width,height){let grid=[]for (let i =0 ; i<height
; i++){ let local=[] for (let j =0 ; j<width ; j++){
local.push({x:j,y:i,isstart:false,istarget:false,weight:1,
iswall:false})
}
grid.push(local)
}
grid[Math.floor(height/2)][Math.floor(width/2)].isstart=tru
e grid[height-2][width-2].istarget=true return grid
}
```

Now we will create state for:

- The mode we're in, either building walls or setting the starting cell.
- The algorithm that we will run.
- The grid which we will equal by default to the grid returned by the function we already created.
- Determining if we're editing or not.
- For the starting and target node coordinates.
- Determining that we want to run the algorithm and clear the grid when changed using use Effect (A separate state for each that, when we change its value, we run a use Effect with an appropriate side effect).

The code will look like this:

```
import { useContext, useState,createContext, useEffect, useRef } from
"react"; import { getGrid } from
"../utils/startinggrid"; const context =
createContext()
export const useParams=()=>{ return
useContext(context)
}
export const ParamsProvider = ({children}) => { const
```

```
[mode,setmode] = useState(null)
const [algo,setalgo] = useState('') const
[run,setrun] = useState(false) const
[grid,setgrid] = useState(getGrid(50,25)) const
[editing,seteditFlag] = useState(false) const
[res,setres] = useState(false) const
start=useRef({x:25,y:12}) const
end=useRef({x:48,y:23})
useEffect(()=>{
restart()
},[res])
function restart(){
setgrid(getGrid(50,25))
}
return (<div>
<context.Provider
value={{mode, setmode, algo,
setalgo, grid, setgrid, setres,
editing, seteditFlag, start,
end, run, setrun, res}}>
{children}
</context.Provider>
</div>)
}
```

And finally, we'll wrap the app component with the ParamsProvider like this:

```
import React from 'react' import
ReactDOM from 'react-dom/client' import
App from './App' import './index.css'
import {ParamsProvider} from './context/context'
ReactDOM.createRoot(document.getElementById('root')).render(
<React.StrictMode>
<ParamsProvider>
<App />
</ParamsProvider>
</React.StrictMode>
)
```

Now to make sure everything is working, import the useParams custom hook to your any components. Now use the console to check its return value. It should return an object with all the variables we've added to the store.

```
import './App.css' import {useParams} from
'./context/context' function App() {
console.log(useParams()) return (
<div></div>)
}
export default App
```

This is how the grid is going to look:

Now we need to add three event listeners to each cell. First, we'll add onMouseDown and onMouseUp event listeners – we use these to set the editing context variable. Then we'll add an onMouseOver which will determine – based on the mode and that editing flag – what changes are being applied to the grid.

We will be updating the grid as usual – we will only use the ref method when running the algorithm. The code will look like this:

```
return (
<div className='board'>
{refarray.map((elem,index)=> { let
classList=['cell'] let
yindex=Math.floor(index/50) let
xindex=index % 50
let cell=grid[yindex][xindex]
if (cell.iswall) { classList.push('wall')
}
return <div key={`${index}`} ref={elem} className={classList.join(' ')}
onMouseDown={()=>{seteditFlag(true)}} onMouseUp={()=> {seteditFlag(false)}}
onMouseMove={()=>{ if (!editing) return const current=
grid[yindex][xindex] if (current.isstart || current.istarget ) return
switch(mode){ case 'setstart':
var newgrid=grid.map((elem)=>{
return elem.map((elem)=>{ if
(!elem.isstart) return elem
return {...elem,isstart:false}
})
})
newgrid[yindex][xindex]={...newgrid[yindex][xindex],isstart:true,istarget:false,
w eight:1,iswall:false} start.current={x:xindex,y:yindex} setgrid(newgrid) break;
case 'settarget':
var newgrid=grid.map((elem)=>{
return elem.map((elem)=>{ if
(!elem.istarget) return elem
return {...elem,istarget:false}
})
})
newgrid[yindex][xindex]={...newgrid[yindex][xindex],isstart:false,istarget:true,
w eight:1,iswall:false} end.current={x:xindex,y:yindex} setgrid(newgrid) break;
case 'addbricks':
var newgrid=grid.slice()
newgrid[yindex][xindex]={...newgrid[yindex][xindex],weight:1,iswall:true}
setgrid(newgrid) break;
case 'addweight':
var newgrid=grid.slice()
newgrid[yindex][xindex]={...newgrid[yindex][xindex],weight:5,iswall:false
}
setgrid(newgrid) break;
default:
return
}}}>
{cell.weight > 1 ? <i className="bi bi-virus"></i> : null}
{cell.isstart ? <i className="bi bi-geo-alt"></i> : null }
{cell.istarget ? <i className="bi bi-geo"></i> : null }
</div>
})}
```

)
If the editing flag is false, we will return. The same applies if the cell is a starting cell or a target one – then we don't want to modify them. Else if the mode is equal to addwalls, then we will modify the corresponding cell in the grid and set the iswall property to true.
Also if the mode is equal to addweight we will modify the corresponding cell in the grid and set the weight property to 5 instead of 1.
For the setstart we will create a copy of the grid where all the cells have the isstart set to false. Then we'll set the corresponding cell for the new start cell to true. The same goes for the settarget mode.
Now you should be able to add walls, weights and change the position of the starting and ending node:
The Algorithms
We can find the shortest path using the algorithms we will implement. Each algorithm finds a path in a unique way and, depending on the algorithm, the output will change.
Let's start with the breath-first search (or BFS) algorithm. We will create a function BFS that takes 5 arguments:
• the graph
• the start and end point coordinates, start and target
• prevmap, which is a hashmap used to track the previous cell for each cell in the grid when the algorithm runs
• hashmap, which is a hashmap that we will use to track visited cells. A hashmap is an object with key value pairs, like a dictionary in Python.
For each cell in the graph, we will create an id x-y which will be unique. We'll set its value to false for hashmap and null for prevmap.
Now we will start with an array with one element – the starting node's coordinates – and a counter set initially to zero. While the length of the array is not zero, we will pop the last element off the array and increment the counter.
Now using the coordinates of the element, we will access its ref and add the visited class with a transition delay proportional to the counter.
Then we will access the siblings of the element from the grid and check if they are visited or not from the hashmap. If they are visited we will ignore them, but if they are not visited we will mark them as visited and add them to the top of the array. Then we'll mark their value in the prevmap to the current element.
When popping elements off, if we come to an element with x and y coordinates equal to those of the target, we will return this object with the current counter.
Depth-first search is very similar: with small changes in the order, we can remove and add elements to the array.
Finally, if there is no path from a to b – for example if a is surrounded by walls – we will return null. This will happen only when the array gets empty before returning a value
We will be running the algorithm only when the run button in the navbar is clicked. That will change the value of run, so we will run a useEffect for this with the context variable run in its dependency array.
Now after pressing the start button this is the output we're going to get:
How to Clear the Board
The side effects of running these algorithms are the classes we've added and the transition delay property (which we need to clear before running the algorithm once

again). That's what we're going to do to reset the grid.
So the final useEffect will iterate over every ref of the refarray and reset its classes and transition delay. Also in the context there is another useEffect that will regenerate a new grid (you can check the context code out). It will look like this:

```
useEffect(()=>{
refarray.forEach((elem)=>{elem.current.style['transition-delay']='0ms'})
refarray.forEach((elem)=>{elem.current.classList.remove('visited');elem.current.c
lassList.remove('path')})
},[res])
```

Now when you click the restart button the board will be back to normal.

Conclusion

Finally in this tutorial we've learned a lot about path finding algorithms, React, contexts, refs, algorithmic thinking and much more. Review 1 1 Excellent Good Satisfactory Approved Not Approved 2 Excellent Good Satisfactory Approved Not Approved 3 Excellent Good Satisfactory Approved Not Approved 4 Excellent Good Satisfactory Approved Not Approved Guide Name & Signature with Date Reviewer Name & Signature with Date Note: All the information regarding performance (Except students' detail) must be filled by the respective Guide. Project/Dissertation Report on PATH FINDING VISUALIZER Submitted in partial fulfillment of the requirement forthe award of the degree of B.TECH. C.S.E. Under the supervision of Name of Supervisior : Dr. Gaurav Sharma Designation : Assistant Proffesor Submitted By Rajat Mittal - 21SCSE1010465 Samir - 21SCSE1010597 Ankita Surbhi - 21SCSE1010941 Himanshu Arya - 21SCSE1010864 SCHOOL OF COMPUTING SCIENCE AND ENGINEERING DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING / DEPARTMENT OF COMPUTERAPPLICATION GALGOTIAS UNIVERSITY, GREATER NOIDA INDIA JUNE, 2023 SCHOOL OF COMPUTING SCIENCE AND ENGINEERING GALGOTIAS UNIVERSITY, GREATER NOIDA CANDIDATE'S DECLARATION I/We hereby certify that the work which is being presented in the project/dissertation, entitled PATH FINDING VISUALIZER in partial fulfillment of the requirements for the award of the B . T E C H . C . S . E . submitted in the School of Computing Science and Engineering of Galgotias University, Greater Noida, is an original work carried out during the period of month, Year to Month and Year, under the supervision of Gaurav Sharma , Assistant Proffesor , Department of Computer Science and Engineering/Computer Application and Information and Science, of School of Computing Science and Engineering , Galgotias University, Greater Noida The matter presented in the project has not been submitted by us for the award of any other degree of this or any other places. Rajat Mittal - 21SCSE1010465 Samir - 21SCSE1010597 Ankita Surbhi - 21SCSE1010941 Himanshu Arya - 21SCSE1010864 This is to certify that the above statement made by the candidates is correct to the best of my knowledge. Dr. Gaurav Sharma Assistant Proffesor CERTIFICATE The Final Thesis/Project/ Dissertation Viva-Voce examination of Rajat Mittal - 21SCSE1010465 Samir - 21SCSE1010597 Ankita Surbhi - 21SCSE1010941 Himanshu Arya - 21SCSE1010864 has been held on 5 th of May , 2023 and his/her work is recommended for the award of B.Tech. C.S.E. Signature of Examiner(s) Signature of Supervisor(s) Signature of Program Chair Signature of Dean Date: June , 2023 Place: Greater Noida ABSTRACT So, here's the thing. Lately, I got pretty interested in web development, but the problem was I have never done it before. Having said that, I am not new to programming. I have worked with languages like C/C++, Java, JavaScript, and my favorite one Python. Python reserves a special place in my heart. Okay, let's get to web development and the project I made and the experience I got making it. My interest in web development grew from the hype and enthusiasm shared by the Vue community. So, I did the basic HTML and CSS so that I can jump right into the land of Vue. And, I have to say that the praise Vue

has got is worth it. It is so easy to delve right into it from the moment you learn it. Thanks to the simplicity built into the language. For me, this is the same aspect because of which Python is so dear to me. Okay, enough of talking. Let me show you what I have built. The project is called a Pathfinding Visualizer, aptly because it does what it says, it finds a path from a source to a destination. This project is based on graph theory. Note: I was inspired to do this project after I went to the website built by Clement Mihailescu. At first, I built a version of this using the Processing framework, but later I learnt web development and built the web version. Algorithm visualization has been high topic in Computer science education for years, but it did not make its way to schools/collages lecture halls as the main educational tool. The present paper identifies two key circumstances that an algorithm visualization must fulfil to be successful: general availability of used software, and visualization of why an algorithm solves the problem rather than what it is doing. One possible method of “why” algorithm visualization is using algorithm unvarying rather than showing the data conversion only. Invariants are known in Program faultless. Keywords : Breadth First Search, Depth First Search, Dijkstra's algorithm , A* algorithms Breadth First Search It is a graph algorithm which does the traversal in a level order fashion. To put it in simple terms, it goes towards the destination layer by layer. It’s like an onion with many layers. You peel out layer by layer to get to the core of the onion. This algorithm gives the shortest path possible. Depth First Search It is another graph traversal algorithm. It uses recursion. This algorithm has preferences for going in a particular direction. To put it simply, let’s say it will go always up until it can’t go further in which case let’s say it will go left and likewise. So is the name depth-first. It’s like taking a piece of big wood. Let’s say I have a wood piece of length 8. At first, I cut it into two halves of 4 each. I take the first half and again cut in into two halves of 2 each. Now I will grab one of my piece with length 2 and cut it into two 1. See how I am always going with the new piece every time. After I get to length 1, I cannot divide it further. So, I will go with the other piece of length 2. After I divide this into two one, I will go with the first piece with length 4 and do the same steps. This algorithm generally does not give the shortest path. Dijkstra's algorithm A common example of a graph-based pathfinding algorithm is Dijkstra's algorithm. This algorithm begins with a start node and an "open set" of candidate nodes. At each step, the node in the open set with the lowest distance from the start is examined. The node is marked "closed", and all nodes adjacent to it are added to the open set if they have not already been examined. This process repeats until a path to the destination has been found. Since the lowest distance nodes are examined first, the first time the destination is found, the path to it will be the shortest path. Dijkstra's algorithm fails if there is a negative edge weight. In the hypothetical situation where Nodes A, B, and C form a connected undirected graph with edges AB = 3, AC = 4, and BC = −2, the optimal path from A to C costs 1, and the optimal path from A to B costs 2. Dijkstra's Algorithm starting from A will first examine B, as that is the closest. It will assign a cost of 3 to it, and mark it closed, meaning that its cost will never be reevaluated. Therefore, Dijkstra's cannot evaluate negative edge weights. However, since for many practical purposes there will never be a negative edgeweight, Dijkstra's algorithm is largely suitable for the purpose of pathfinding. A* algorithm A* is a variant of Dijkstra's algorithm commonly used in games. A* assigns a weight to each open node equal to the weight of the edge to that node plus the approximate distance between that node and the finish. This approximate distance is found by the heuristic, and represents a minimum possible distance between that node and the end. This allows it to eliminate longer paths once an initial path is found. If there is a path of length x between the start and finish, and the minimum distance between a node and the finish is greater than x, that node need not be examined. A* uses this heuristic to improve on the behavior relative to Dijkstra's algorithm. When the heuristic evaluates to zero, A* is equivalent to Dijkstra's algorithm. As the heuristic estimate increases and gets closer to the

true distance, A* continues to find optimal paths, but runs faster (by virtue of examining fewer nodes). When the value of the heuristic is exactly the true distance, A* examines the fewest nodes. (However, it is generally impractical to write a heuristic function that always computes the true distance, as the same comparison result can often be reached using simpler calculations – for example, using Manhattan distance over Euclidean distance in two-dimensional space.) As the value of the heuristic increases, A* examines fewer nodes but no longer guarantees an optimal path. In many applications (such as video games) this is acceptable and even desirable, in order to keep the algorithm running quickly. EXISTING PROBLEMS In today's time there are multiple routes of travelling to a particular destination but we don't know the most time sufficient route. This problem has still not been solved which can cause a lot of time management crisis and can also lead to wastage of fuels which hence makes it very cost consuming. Since not everywhere there is internet connectivity so once we go offline, we lose our destination path which can create a lot of obvious problems. Ever since the emergence of the first primitive search algorithms, there has been a solution to the most basic definition of the pathfinding problem, in which there are no spatial or temporal constraints on the solution. Given enough time and memory, even the most basic of brute-force search algorithms can exhaustively search the solution space to find the optimal path between two points. This definition of the problem is, however, very optimistic. Computers are, by the laws of physics, no Turing Machines, and there is, and will always be, an upper bound on how much memory they can facilitate; thus all pathfinding solutions have spatial constraints. The temporal constraint comes from the simple fact that there is often a querying system that has requested the path for one reason, or another, and can't wait indefinitely for an answer. There are three groups of constraints of interest to this research: • Temporal: the time it takes a pathfinding solution to find the most optimal path it can generate. • Spatial: the amount of memory that must be allocated by the pathfinding solution during runtime. • Path Optimality: the degree to which the resulting path corresponds to the shortest path possible in a Newtonian universe. On a Nintendo DS, the spatial constraints can be considered very hard. All pathfinding algorithms need some form of representation of the world they are supposed to find paths in; a simplified abstraction of the world, which the algorithms can understand. The memory architecture of the DS2 poses great limitations on how much information a world abstraction can contain. Memory available for dynamic allocation during runtime is, for the same reasons, very restricted. In any video game, responsiveness is of great importance. The speed of the pathfinding algorithm used in an RTS game can greatly affect how responsive the game feels to the user; when a unit is ordered to move, it should, ideally, do so without any noticeable delay. To further add to the temporal constraints, pathfinding is seldom the only concurrent task of a video game, and is often given a small percentage of the processing time, in favor of graphics- and sound processing, and other systems of the game. Considering this, and with several units requesting path information simultaneously being a common thing in most RTS games, the temporal constraints of pathfinding in an RTS game can be viewed as hard. The path optimality constraint of an RTS game can be considered to be moderate. If a path exists between two points, the pathfinder must generate a result, but not necessarily the shortest possible. As long as the movements of units aren't prone to peculiarities, or can be viewed as moronic by the player, close enough is often accepted as good enough. When combining the constraints of an RTS game with those generated by the DS, the spatial- and path optimality constraints are transferred largely unchanged. The temporality constraint must, however, be modified, as the limitations of the DS influence the performance of the algorithm. By itself, the DS doesn't generate any temporal constraints, as there are no requirements on how long time the system can work before it must generate an answer. But, as previously mentioned, much attention must be given the responsiveness of an RTS, and the limitations of the DS make the need for a fast algorithm somewhat more vital

than on a modern PC3; making the temporal constraints very hard. PROPOSED SOLUTIONS Our visualizer will help solve all the above-mentioned problems. It will give the user most time and cost sufficient, i.e., the shortest route to reach their desired destination. Even if the networks fail the user can still access the map with the help of our offline feature which supports the satellite view as well as the road view in order to reach his/her desired destination. The offline feature would come in very handy for elderly people who sometimes may face issues while using a smartphone with internet connectivity. In sorting algorithms, the user has to give how many inputs and the set of data for inputs or he/she can generate random array inputs using the Generate Button. Then select the particular algorithm from the list and then the visualization of the selected algorithm is shown with the given inputs. Also, the web-based platform has the feature to visualize the algorithm as per the user's need. The proposed system involves the simulation of the different type of algorithms codes. As you can see, there are no major components besides the three coding languages. Most websites have tools or scripts that require a server on the back-end (like PHP), but it is not necessary in this case since React JS runs right in the user's browser. HTML5 and CSS are used for the interface. The HTML5 communicates with the React JS code and vice versa to launch the appropriate algorithms and update the interface accordingly, as seen with a single, bidirectional arrow. As the React JS was modified from a functional programming focus to a more object-oriented one, the parts of the HTML5 that did change were the function calls for each button. All of the back-end interaction is abstracted to the various buttons for selecting algorithms and running the animation. TOOLS AND TECHNOLOGY USED An efficient algorithm is one that calculates the shortest path with the fewest number of node visitations. Among all the Pathfinding Algorithms, Dijkstra's Algorithms is least efficient as it has no method of cutting down on search space and it made far more node visitations during each path calculation phase. Depending, on the situation A* and D* are the most efficient pathfinding Algorithms We are going to build this website with the help of HTML(frontend) and Javascript (backend). ALGORITHM USED: Sorting Visualization(bubble sort, insertion sort, Quick sort) Path Finding Maze breadth RESULTS AND OUTPUT The goal of the sorting visualizer tool is to make use of bars and their comparisons to help understand sorting algorithms. This is user flow which shows how sorting visualizer tools can be used. CONCLUSION AND FUTURE SCOPE Thus, with our initiative we aim to create to a software which will save time and money and help the user in reaching their destination as soon as possible. In future we hope to create an offline app which will have pre-saved routes to popular destinations. With the shift of remote and digital literacy, a combined platform serving the effective literacy requirements of students is required. Algorithm Visualizer is a combined platform that's a comprehensive result for educators and students to educate and learn online effectively. It substantially focuses on "algorithm visualization", which allows a better understanding of its inflow and operation. The developed platform offers a complete perspective of visualizations for sorting and pathfinding algorithms so far. Also, the web-based platform can run on small devices while providing the feature to visualize at one's own pace. As a future scope, more Algorithms for Trees, Graphs, Linked List and many more can be added. To empower learning of the learner concept of community learning through forums, discussion and user-based feedback can be added. The objective of the platform is to reduce the fear level in the minds of learners especially students regarding Algorithms and Data Structures. Project Setup For this tutorial I will be using Vite, which is a tool that helps you start projects way faster than npm create-react-app. First, install Vite if you haven't done that yet. Create a folder in a well-known directory on your machine. Then using the terminal, navigate to this directory and run these commands: $ cd my-app $ npm install Now inside the src folder, create a components folder, a contexts folder, and a utils folder – and you're done. Here's a screenshot of what your project should look like: Add Bootstrap Now we need to

add bootstrap to our project for the buttons and icons we're going to add, since we need to focus on the JavaScript part. Represent the Cells For the representation of the grid and the cells, I will be building a two dimensional array of objects containing all the properties we need to represent a certain cell. Now in the utils folder, create a startingGrid.js file. Here, you'll write a function that returns a grid which is a two dimensional array of objects representing the cells. Each object will have these properties: x, y, isstarting, istarget, iswall, and weight. x and y represent the coordinates of the cell. isstarting is a Boolean that's only true for the starting cell. istarget is similar, but for the target node. Is wall is a Boolean that's only true for walls. And weight is an integer. All normal cells have a weight of 1, and the weighted cells have a weight of 5. The function should look like this: exportfunction getGrid(width,height){let grid=[]for (let i =0 ; i{ return useContext(context) } export const ParamsProvider = ({children}) => { const [mode,setmode] = useState(null) const [algo,setalgo] = useState('') const [run,setrun] = useState(false) const [grid,setgrid] = useState(getGrid(50,25)) const [editing,seteditFlag] = useState(false) const [res,setres] = useState(false) const start=useRef({x:25,y:12}) const end=useRef({x:48,y:23}) useEffect(()=>{ restart() },[res]) function restart(){ setgrid(getGrid(50,25)) } return ({children}
) } And finally, we'll wrap the app component with the ParamsProvider like this: import React from 'react' import ReactDOM from 'react-dom/client' import App from './App' import './index.css' import {ParamsProvider} from './context/context' ReactDOM.createRoot(document.getElementById('root')).render( ) Now to make sure everything is working, import the useParams custom hook to your any components. Now use the console to check its return value. It should return an object with all the variables we've added to the store. import './App.css' import {useParams} from './context/context' function App() { console.log(useParams()) return (
) } export default App This is how the grid is going to look: Now we need to add three event listeners to each cell. First, we'll add onMouseDown and onMouseUp event listeners – we use these to set the editing context variable. Then we'll add an onMouseOver which will determine – based on the mode and that editing flag – what changes are being applied to the grid. We will be updating the grid as usual – we will only use the ref method when running the algorithm. The code will look like this: return (
{refarray.map((elem,index)=> { let classList=['cell'] let yindex=Math.floor(index/50) let xindex=index % 50 let cell=grid[yindex][xindex] if (cell.iswall) { classList.push('wall') } return
{seteditFlag(true)}} onMouseUp={()=> {seteditFlag(false)}} onMouseMove={()=>{ if (!editing) return const current= grid[yindex][xindex] if (current.isstart || current.istarget) return switch(mode){ case 'setstart': var newgrid=grid.map((elem)=>{ return elem.map((elem)=>{ if (!elem.isstart) return elem return {...elem,isstart:false} }) }) newgrid[yindex][xindex]={...newgrid[yindex][xindex],isstart:true,istarget:false, w eight:1,iswall:false} start.current={x:xindex,y:yindex} setgrid(newgrid) break; case 'settarget': var newgrid=grid.map((elem)=>{ return elem.map((elem)=>{ if (!elem.istarget) return elem return {...elem,istarget:false} }) }) newgrid[yindex][xindex]={...newgrid[yindex][xindex],isstart:false,istarget:true, w eight:1,iswall:false} end.current={x:xindex,y:yindex} setgrid(newgrid) break; case 'addbricks': var newgrid=grid.slice() newgrid[yindex][xindex]={...newgrid[yindex][xindex],weight:1,iswall:true} setgrid(newgrid) break; case 'addweight': var newgrid=grid.slice() newgrid[yindex][xindex]={...newgrid[yindex][xindex],weight:5,iswall:false } setgrid(newgrid) break; default: return }}}> {cell.weight > 1 ? : null} {cell.isstart ? : null } {cell.istarget ? : null }

})}
) If the editing flag is false, we will return. The same applies if the cell is a starting cell or a target one – then we don't want to modify them. Else if the mode is equal to addwalls, then we will modify the corresponding cell in the grid and set the iswall property to true. Also if the mode is equal to addweight we will modify the corresponding cell in the grid and set the weight property to 5 instead of 1. For the setstart we will create a copy of the grid where all the cells have the isstart set to false. Then we'll set the corresponding cell for the new start cell to true. The same goes for the settarget mode. Now you should be able to add walls, weights and change the position of the starting and ending node: The Algorithms We can find the shortest path using the algorithms we will implement. Each algorithm finds a path in a unique way and, depending on the algorithm, the output will change. Let's start with the breath-first search (or BFS) algorithm. We will create a function BFS that takes 5 arguments: • the graph • the start and end point coordinates, start and target • prevmap, which is a hashmap used to track the previous cell for each cell in the grid when the algorithm runs • hashmap, which is a hashmap that we will use to track visited cells. A hashmap is an object with key value pairs, like a dictionary in Python. For each cell in the graph, we will create an id x-y which will be unique. We'll set its value to false for hashmap and null for prevmap. Now we will start with an array with one element – the starting node's coordinates – and a counter set initially to zero. While the length of the array is not zero, we will pop the last element off the array and increment the counter. Now using the coordinates of the element, we will access its ref and add the visited class with a transition delay proportional to the counter. Then we will access the siblings of the element from the grid and check if they are visited or not from the hashmap. If they are visited we will ignore them, but if they are not visited we will mark them as visited and add them to the top of the array. Then we'll mark their value in the prevmap to the current element. When popping elements off, if we come to an element with x and y coordinates equal to those of the target, we will return this object with the current counter. Depth-first search is very similar: with small changes in the order, we can remove and add elements to the array. Finally, if there is no path from a to b – for example if a is surrounded by walls – we will return null. This will happen only when the array gets empty before returning a value. The code will look like this: function BFS(graph,hashmap,prevmap,start,target){ let queue=[start] let count=0 hashmap[`${start.x}-${start.y}`]=true while (queue.length > 0){ count+=1 let c=queue.pop() refarray[c.x+c.y*50].current.style['transition-delay']=`${count * 8}ms` refarray[c.x+c.y*50].current.classList.add('visited') if (c.x == target.x && c.y == target.y) return [c,count] if(c.x+1 < 50 && !hashmap[`${c.x+1}-${c.y}`] && !graph[c.y][c.x+1].iswall){ queue.unshift({x:c.x +1,y:c.y}) prevmap[`${c.x+1}-${c.y}`]={...c} hashmap[`${c.x+1}-${c.y}`]=true } if(c.x-1 >=0 && !hashmap[`${c.x-1}-${c.y}`] && !graph[c.y][c.x-1].iswall){ queue.unshift({x:c.x -1,y:c.y}) prevmap[`${c.x-1}-${c.y}`]={...c} hashmap[`${c.x-1}-${c.y}`]=true } if(c.y+1 < 25 && !hashmap[`${c.x}-${c.y+1}`] && !graph[c.y+1][c.x].iswall){ queue.unshift({x:c.x ,y:c.y+1}) prevmap[`${c.x}-${c.y+1}`]={...c} hashmap[`${c.x}-${c.y+1}`]=true } if(c.y-1 >=0 && !hashmap[`${c.x}-${c.y-1}`] && !graph[c.y-1][c.x].iswall){ queue.unshift({x:c.x ,y:c.y-1}) prevmap[`${c.x}-${c.y-1}`]={...c} hashmap[`${c.x}-${c.y-1}`]=true } } return null } function BDS(graph,hashmap,prevmap,start,target){ let queue=[start] let count=0 hashmap[`${start.x}-${start.y}`]=true while (queue.length > 0){ count+=1 let c=queue[0] queue.shift() refarray[c.x+c.y*50].current.style['transition-delay']=`${count * 8}ms` refarray[c.x+c.y*50].current.classList.add('visited') if (c.x == target.x && c.y == target.y)

return [c,count] if(c.y+1 < 25 && !hashmap[`${c.x}-${c.y+1}`] && !graph[c.y+1][c.x].iswall){ queue.unshift({x:c.x ,y:c.y+1}) prevmap[`${c.x}-${c.y+1}`]={...c} hashmap[`${c.x}- ${c.y+1}`]=true } if(c.x-1 >=0 && !hashmap[`${c.x-1}-${c.y}`] && !graph[c.y][c.x-1].iswall){ queue.unshift({x:c.x -1,y:c.y}) prevmap[`${c.x-1}-${c.y}`]={...c} hashmap[`${c.x-1}-${c.y}`]=true } if(c.y-1 >=0 && !hashmap[`${c.x}-${c.y-1}`] && !graph[c.y-1][c.x].iswall){ queue.unshift({x:c.x ,y:c.y-1}) prevmap[`${c.x}-${c.y-1}`]={...c} hashmap[`${c.x}-${c.y-1}`]=true } if(c.x+1 < 50 && !hashmap[`${c.x+1}-${c.y}`] && !graph[c.y][c.x+1].iswall){ queue.unshift({x:c.x +1,y:c.y}) prevmap[`${c.x+1}-${c.y}`]={...c} hashmap[`${c.x+1}-${c.y}`]=true } } return null } We will be running the algorithm only when the run button in the navbar is clicked. That will change the value of run, so we will run a useEffect for this with the context variable run in its dependency array. Now after pressing the start button this is the output we're going to get: How to Clear the Board The side effects of running these algorithms are the classes we've added and the transition delay property (which we need to clear before running the algorithm once again). That's what we're going to do to reset the grid. So the final useEffect will iterate over every ref of the refarray and reset its classes and transition delay. Also in the context there is another useEffect that will regenerate a new grid (you can check the context code out). It will look like this: useEffect(()=>{ refarray.forEach((elem)=>{elem.current.style['transition-delay']='0ms'})
refarray.forEach((elem)=>{elem.current.classList.remove('visited');elem.current.c
lassList.remove('path')}) },[res]) Now when you click the restart button the board will be back to normal. Conclusion Finally in this tutorial we've learned a lot about path finding algorithms, React, contexts, refs, algorithmic thinking and much more.