Path Optimization of UCLA for the Physically Disabled

Hamza Khan | Jason Liu | Ryan Liu | Shaash Sivakumar | Marc Walden

Problem Description

- Physically Disabled Students comprise 2.1% of UCLA student body (almost 1000 students)
- Campus constructed with steep slopes and stairs to account for dramatic changes in elevation
- Accessibility issues for disabled students, as current routes are very inefficient and inconvenient
- Wheel transportation including skateboards and electric scooters are also inconvenient

Goals of the Model

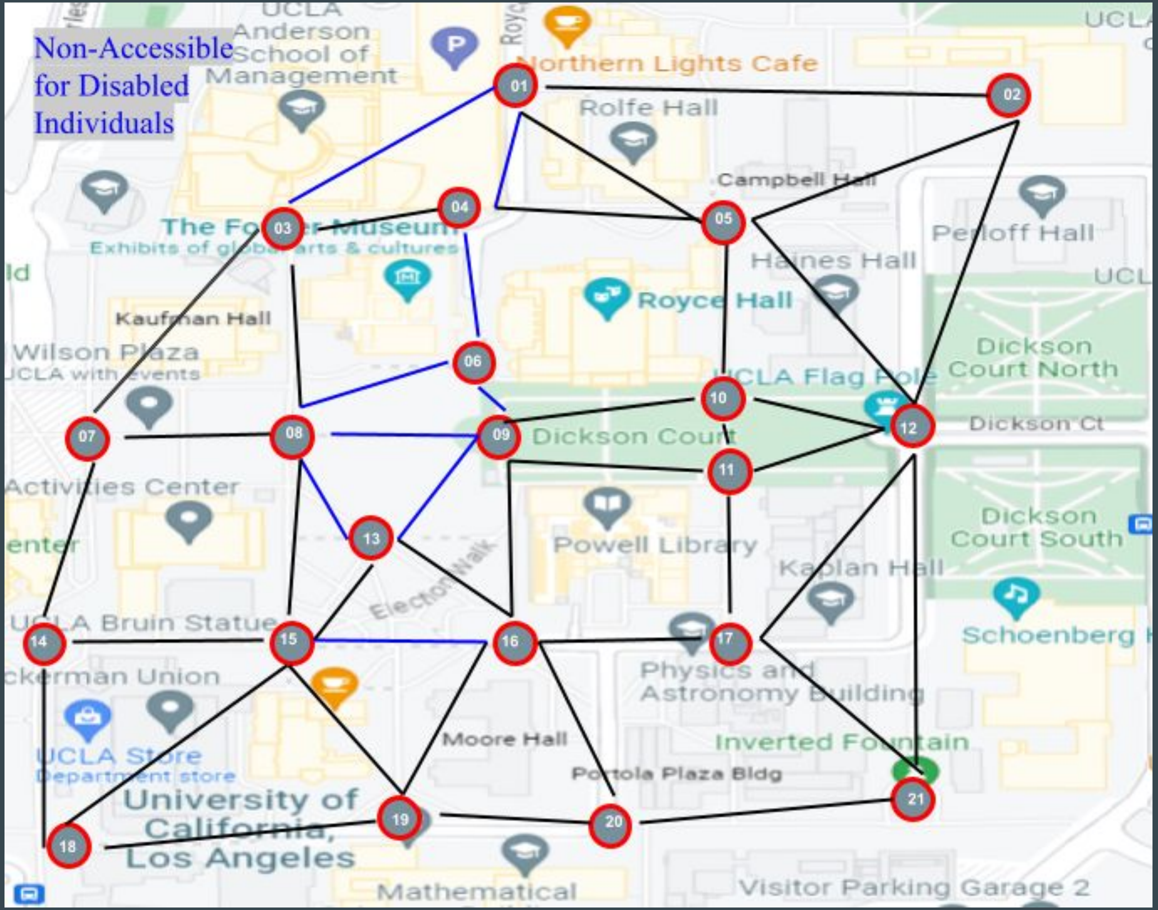
- Identify least accessible areas on campus and propose efficient alternatives
- Simulate a physically disabled individual and a control non-disabled individual walking different routes of campus
- Use path optimization algorithms to produce the most efficient route to a given destination
- Compare the mean travel times between simulations and discuss why paths are inefficient
- Propose solutions to inefficient paths

Simplifications and Assumptions

- Confining model to a rectangle of UCLA with Bunche Hall, Anderson School of Management, Ackerman Turnaround, and Pritzker Hall denoting the four corners
- Use strategically selected notes that encompass the regions possible paths
- Use manual wheelchair user as the physically disabled simulation
- Assume a pace of 2 mph for wheelchair user and 3 mph for control regardless of fatigue/length of travel
- Assume natural gradient of deceleration for inclines
- Assume inclines above 35 degrees are inaccessible

Modeling (Graph Theory)

- **Nodes**: Selected 21 key intersections on campus
- **Paths**: All existing connections between 2 nodes



Data Collection

- Simulated average walking and wheelchair speed to traverse through all possible paths, collected time
- Assumed 2 mph for disabled speed, 3 mph for non-disabled speed
- Assume natural gradient of pace decrease when walking up slope, also any route that requires above 35 degree incline is inaccessible for wheelchair
- Recorded times (in seconds, nearest hundredth) as the weight for each path

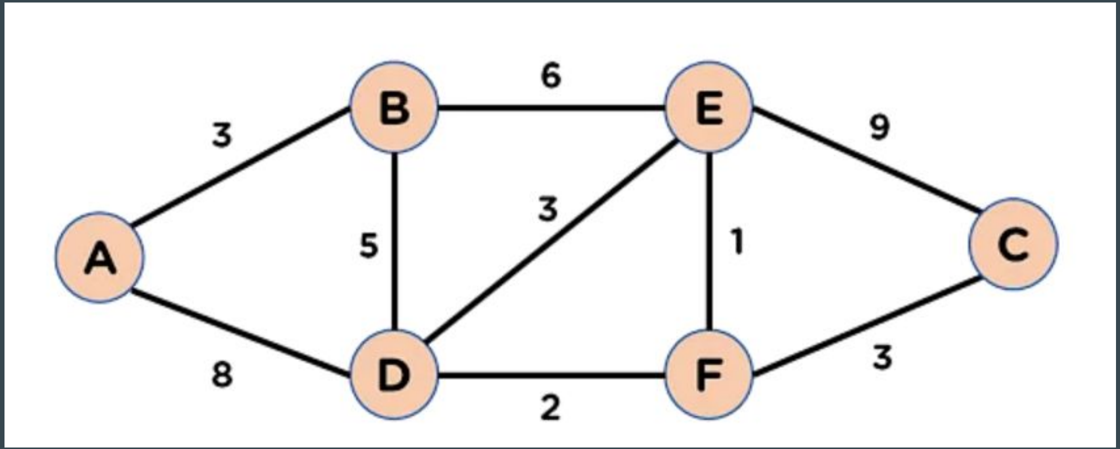
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1		97.28	205.66	51.28																	
2	85.34				106.75							169.41									
3	216.58			107.85			69.19	109.4													
4	59.84		92.34		110.39	105.37															
5		114.99		110.39						63.85		106.19									
6				97.41				93.97	130.63												
7			69.19					65.7						92.56							
8			109.4			93.97	65.7		101.49				64.58		113.4						
9						107.52		72.93		56.71	56.71		60.31			126.62					
10					63.85				56.71		30.34	43.33									
11									56.71	30.34		42.21					63.85				
12		171.38			106.19					43.33	42.21						110.97				205.89
13								54.25	63.55						68.35	101.93					
14							92.56								69.17			79.06			
15								115.71					91.41	59.17		98.7		194.01	133.45		
16									128.96				79.9		66.57		48.85		105.85	79.06	
17											63.85	111.37				45.85					88.11
18														79.85	179.74				86.7		
19															133.23	110.85		86.7		54.82	
20																79.06			43.93		145.41
21												212.11					88.11			145.41	

0: Non-disabled

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1		117.03			163.37				117.03			163.37				117.03			163.37		
2	108.13				200.43							283.96									
3				168.32			112.24	122.13													
4			167.36		86.58																
5	163.37	227.27		85.27						128.57		184.12									
6																					
7			127.08					71.44						107.12							
8			131.13				65.34								116.04						
9										161.04	152.49					98.94					
10					135.27				163.29		37.9	74.43									
11									159.16	37.9		74.43					199.99				
12		293.94			179.18					74.43	74.43						214.42				248.45
13															57.87	215.95					
14							108.89								110.04			87.29			
15								113.24					58.3	105.72					252.12		
16									98.94				191.72				50.35		173.62	95.17	
17											133.29	207.71				50.35					217.93
18														90.46							
19															239.1	196.67				108.65	
20																88.02			90.41		168.63
21												243.45					222.3			140.79	

1: Disabled

Method 1 - Dijkstra's Algorithm



Objective: To find the minimum time from starting node to ending node

Process: To find the time it takes to go from A to B for example, compare the time from A to B with the time it takes to go from A to all the nodes in between A and B and then from that node to B. The smallest of these times replaces our value from A to B.

To find the time it takes to go from A to E, find all the nodes in between A and E, (B and D) and then use those paths to find the quickest path.

End > v Start	A	B	C	D	E	F
A	0	3	∞	8	∞	∞
B	3	0	∞	5	6	∞
C	3	∞	0	∞	9	3
D	8	5	∞	0	3	2
E	∞	6	9	3	0	1
F	∞	∞	3	2	1	0

End > v Start	A	B	C	D	E	F
A	0	3	13	8	9	10
B	3	0	10	5	6	7
C	13	10	0	5	4	3
D	8	5	5	0	3	2
E	9	6	4	3	0	1
F	10	7	3	2	1	0

Code Snippet

```python
def connected_node_times(start, end, num):
  if(start == end):
    return 0
  direct_time = datas[num][start][end]
  between_nodes = []
  length = 0
  for i in range(0, 21):
    vec = [i != start, i != end, math.isinf(datas[num][start][i]) == False, math.isinf(datas[num][i][end]) == False, i != j]
    if all(item == True for item in vec):
      between_nodes.append(i)
      length = length + 1
  indirect_times = [0 for _ in range(length)]
  for i in range(0, length):
    indirect_times[i] = datas[num][start][between_nodes[i]] + datas[num][between_nodes[i]][end]
  if math.isinf(datas[num][start][end]) == True:
    if len(indirect_times) > 0:
      return min(indirect_times)
    else:
      return 9999
  else:
    allthetimes = [0 for _ in range(length + 1)]
    allthetimes[0] = direct_time
    for k in range(0, length):
      allthetimes[k + 1] = indirect_times[k]
    if len(allthetimes) > 0:
      return min(allthetimes)
    else:
      return 9999
def get_Time(start, end, num):
  mat = datas[num]
  counter = 0
  change = True
  while change:
    change = False
    counter = counter + 1
    for i in range(0, 21):
      for j in range(0, 21):
        if connected_node_times(i, j, num) != 9999 and connected_node_times(i, j, num) < datas[num][i][j]:
          datas[num][i][j] = connected_node_times(i, j, num)
          change = True
  return datas[num][start - 1][end - 1]
```

datas is a matrix that contains all the times, where the row number represents the start-node and the column number represents the end-node

Connected_node_times calculates the minimum time required to go from start-node to finish-node, if the start-node and finish-node are either a direct path or if they have one node in between. If that's not the case, it returns 9999 (representing infinity).

get_Time calculates the minimum time to go from any start-node to any end-node. It keeps updating the minimum time to go from any start-node to any end-node until the matrix stores the minimum time for all paths. Then, it accesses the time from the matrix

Method 2 - Least Resistance Algorithm

- Analog- electricity arcing through a non uniform material
- Steadily increase Voltage: amount of weight the algorithm allows current paths to overcome (ignoring initial values)
- Construct & maintain two arrays, representing possible paths that can be achieved from start and end nodes (respectively), within the current voltage (end paths are reversed times)
- Algorithm iteratively explores all possible paths for both these arrays, building a collection of paths as the voltage increases
- Eventually, paths converge (either discretely or continuously), representing the shortest route to get from start to end

Code Breakdown

- ❖ Initial Values for Path Arrays: Searches for any nodes adjacent to start and end
- ❖ Iterates through existing start array paths, if voltage allows it to, adds any extensions as a new path (non duplicate)
- ❖ Same thing implemented for ending array paths (not shown)

```python
def least_resistance(start, end, d = 0):
  # breaks if already at the end
  if start == end:
    return [0, [start]]
  # sets default values, giving all adjoining nodes to start and end
  completed = False
  spaths = []
  for i in range(1,22):
    if data[d][start-1][i-1] != "":
      spaths.append([i])
  epaths = []
  for i in range(1,22):
    if data[d][i-1][end-1] != "":
      epaths.append([i])
  voltage = 0
  route = []
  # main loop, increases voltage and runs a series of checks
  while not completed:
    voltage += 1
    # for each set of paths, calculates residual voltage left over after subtracting resistance of each existing path
    for paths in spaths:
      residual = voltage
      for i in range(len(paths)):
        if i == 0:
          residual -= float(data[d][start-1][paths[0]-1])
        else:
          residual -= float(data[d][paths[i-1]-1][paths[i]-1])
      # checks if the residual is sufficient to complete another path
      for i in range(1, 22):
        if i != start and paths.count(i) == 0:
          if data[d][paths[len(paths)-1]-1][i-1] != "":
            if residual >= float(data[d][paths[len(paths)-1]-1][i-1]):
              # appends the path as long as it isn't a duplicate
              new_path = []
              for element in paths:
                new_path.append(element)
              new_path.append(i)
              duplicate = False
              for expaths in spaths:
                if expaths == new_path:
                  duplicate = True
              if not duplicate:
                spaths.append(new_path)
```

- ❖ Discrete: Use boolean checklist to check if paths end in the same node
- ❖ Continuous: If paths end in connecting nodes, check to see if remaining voltage is enough to traverse between the nodes

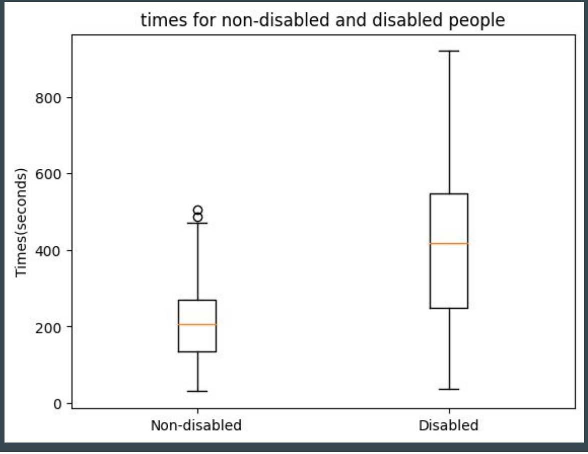
```python
    # check condition for start and end paths meeting on a path
    for paths_s in spaths:
      # finds the residual of all possible start and end paths
      if not completed:
        sresidual = voltage
        for i in range(len(paths_s)):
          if i == 0:
            sresidual -= float(data[d][start-1][paths_s[0]-1])
          else:
            sresidual -= float(data[d][paths_s[i-1]-1][paths_s[i]-1])
        for paths_e in epaths:
          eresidual = voltage
          for j in range(len(paths_e)):
            if j == 0:
              eresidual -= float(data[d][paths_e[0]-1][end-1])
            else:
              eresidual -= float(data[d][paths_e[j]-1][paths_e[j-1]-1])
          # checks if the end nodes are adjoining and sum of residuals can span it
          if data[d][paths_s[len(paths_s)-1]-1][paths_e[len(paths_e)-1]-1] != "":
            if sresidual + eresidual > float(data[d][paths_s[len(paths_s)-1]-1][paths_e[len(paths_e)-1]-1]):
              # creates the route by splicing together the start and reversed end path
              route.append(start)
              for element in paths_s:
                route.append(element)
              for k in range(len(paths_e)-1, -1, -1):
                route.append(paths_e[k])
              route.append(end)
              completed = True
              break
```

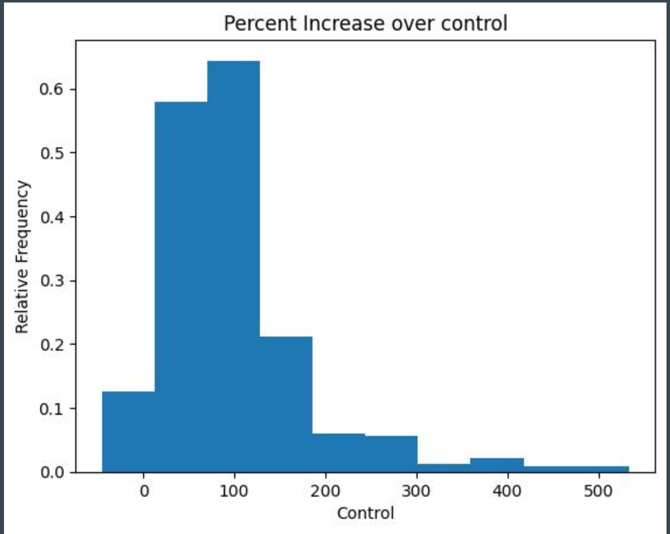
- ❖ Route constructed by reordering and combining paths
- ❖ Accurate time (0.01) found by adding the weights of each route pair

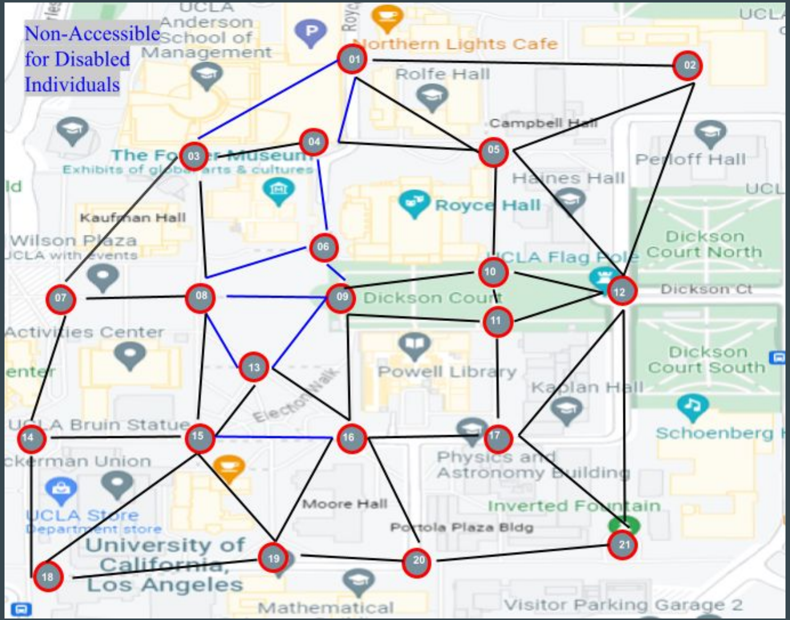
Results

- ❖ 98% accuracy for least resistance algorithm
- ❖ Based on the plot, average times for disabled is much higher than non-disabled
- ❖ Mean times: 211 vs 390 (+85%)
- ❖ Longest Routes Mostly Involved:
  - ➢ 15-19: extremely inefficient winding slope
  - ➢ 8-9: can't use stairs, need to use the winding slope again
  - ➢ 4-1: indirect, requires u-turn
- ❖ Possible Solutions:
  - ➢ 15-19: build exterior elevator. (90s)
    Mean time: 374
  - ➢ 8-9: create ramp next to steps. (270s, 240s)
    Mean time: 383
  - ➢ 4-1: create route through/around. (160s)
    Mean time: 385

Mean time with all 3 solutions: 362






Non-Accessible
for Disabled
Individuals
UCLA Anderson School of Management
Northern Lights Cafe
Rolfe Hall
Campbell Hall
Perloff Hall
Haines Hall
Royce Hall
The Fowler Museum
Exhibits of global arts & cultures
Kaufman Hall
Wilson Plaza
UCLA with events
UCLA Flag Pole
Dickson Court North
Dickson Ct
Dickson Court
Dickson Court South
Activities Center
Powell Library
Kaplan Hall
Schoenberg
UCLA Bruin Statue
Ackerman Union
Election Walk
Physics and Astronomy Building
Inverted Fountain
UCLA Store
Department store
Moore Hall
Portola Plaza Bldg
University of California, Los Angeles
Mathematical
Visitor Parking Garage 2
01
02
03
04
05
06
07
08
09
10
11
12
13
14
15
16
17
18
19
20
21

Future Improvements

- ❖ Improve accuracy of least resistance model
- ❖ Possible expansion to more areas, perhaps even areas of Los Angeles
- ❖ Database expansion, account for more areas of UCLA
  - ➢ Include indoor or smaller paths
  - ➢ Automation in data collection (Google Maps API Calls)
- ❖ Expand to include different modes of transport
- ❖ Include stochastic factors, such as congestion and traffic signals

References

- Kapoor, Anmol. “What Is Dijkstra’s Algorithm? Here’s How to Implement It with Example?” Simplilearn.Com, 21 Feb. 2023, www.simplilearn.com/tutorials/cyber-security-tutorial/what-is-dijkstras-algorithm.
- Maverick, John. “How Fast Can a Manual Wheelchair Go: Helpful Guide 2023.” Wheelchairsvilly.Com, 24 Jan. 2023, wheelchairsvilly.com/how-fast-can-a-manual-wheelchair-go/.
- Metterhausen, Fred. “Elevation Calculator: Find Your Current Elevation, an Address, or a Point on the Map.” Elevation Calculator: Find My Elevation on a Map., www.mapdevelopers.com/elevation_calculator.php. Accessed 13 June 2023.
- Systems, UCLA Facilities Management Information. Map.Ucla.Edu, map.ucla.edu/. Accessed 13 June 2023.
- Wilson, Robin J. Introduction to Graph Theory. Prentice Hall, 2015.

Thank You!