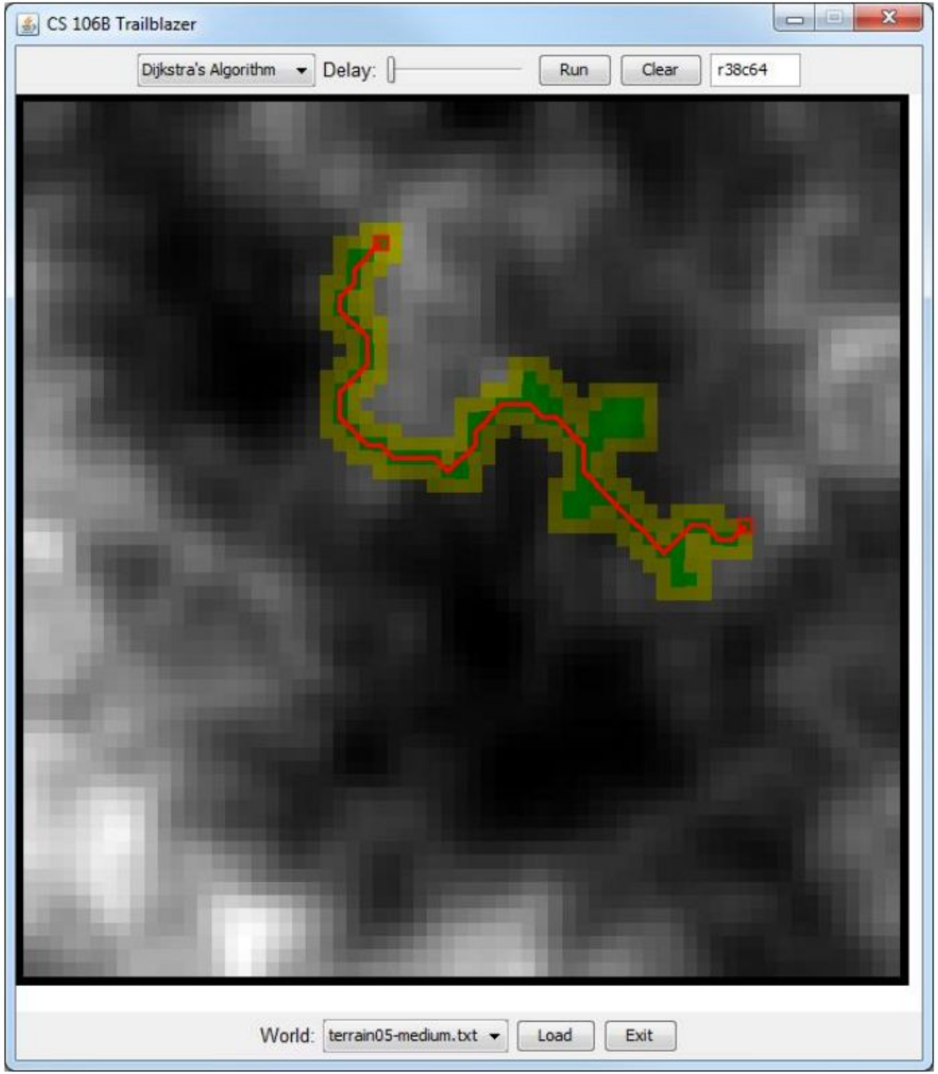
Trailblazer YEAH Hours

Alexander De Baets


CS 106B Trailblazer
Dijkstra's Algorithm
Delay:
Run
Clear
r38c64
World:
terrain05-medium.txt
Load
Exit

What do you have to do?

- Vector depthFirstSearch(BasicGraph& graph, Vertex* start, Vertex* end)
- Vector breadthFirstSearch(BasicGraph& graph, Vertex* start, Vertex* end)
- Vector dijkstrasAlgorithm(BasicGraph& graph, Vertex* start, Vertex* end)
- Vector aStar(BasicGraph& graph, Vertex* start, Vertex* end)
- Set kruskal(BasicGraph& graph)

All your code should live in trailblazer.cpp, don't modify any other files!

Searching

You'll be given:

- Start Vertex
- End Vertex
- BasicGraph

Your job is to find a path from Start Vertex to End Vertex.

What does the Vertex look like?

`string` **`name`**	vertex's name, such as `"r34c25"` or `"vertex17"`
`Set<Edge*>` **`edges`**	edges outbound from this vertex
`double` **`cost`**	cost to reach this vertex (initially `0`)
`bool` **`visited`**	whether this vertex has been visited yet (initially `false`)
`Vertex*` **`previous`**	pointer to a vertex that comes before this one; initially `NULL`
`void` **`setColor`**`(Color c)`	sets this vertex to be drawn in the given color in the GUI, one of `WHITE`, `GRAY`, `YELLOW`, or `GREEN`
`Color` **`getColor`**`()`	returns color you set previously using `setColor`; initially `UNCOLORED`
`void` **`resetData`**`()`	sets `cost`, `visited`, `previous`, and color back to their initial values
`string` **`toString`**`()`	returns a printable string representation of the vertex for debugging

What does the Edge look like?

Edge member	Description
`Vertex* start`	the starting vertex of this edge
`Vertex* finish`	the ending vertex of this edge (i.e., `finish` is a neighbor of `start`)
`double cost`	cost to traverse this edge

Section Seven!

The first two pages are just HUGE reference sheets! You should go through it before you start this assignment, it will save you a lot of pain!

Depth-first search (DFS) pseudo-code:

```
function dfs(v1, v2):
  dfs(v1, v2, { }).

function dfs(v1, v2, path):
  path += v1.
  mark v1 as visited.
  if v1 is v2:
    a path is found!

  for each unvisited neighbor n of v1:
    if dfs(n, v2, path) finds a path:
      a path is found!

  path -= v1.  // path is not found.
```

Breadth-first search (BFS) pseudo-code:

```
function bfs(v1, v2):
  queue := {v1}.
  mark v1 as visited.

  while queue is not empty:
    v := queue.dequeue().
    if v is v2:
      a path is found!

    for each unvisited neighbor n of v:
      mark n as visited.
      queue.enqueue(n).

  // path is not found.
```

Dijkstra's algorithm pseudo-code:

```
function dijkstra(v1, v2):
  for each vertex v:
    v's cost := infinity.
    v's previous := none.
  v1's cost := 0.
  pqueue := {v1, at priority 0}.

  while pqueue is not empty:
    v := pqueue.dequeue().
    mark v as visited.
    for each unvisited neighbor n of v:
      cost := v's cost +
              weight of edge (v, n).
      if cost < n's cost:
        n's cost := cost.
        n's previous := v.
        enqueue/update n in pqueue.
  reconstruct path back from v2 to v1.
```

A* algorithm pseudo-code:

```
function astar(v1, v2):
  for each vertex v:
    v's cost := infinity.
    v's previous := none.
  v1's cost := 0.
  pqueue := {v1, at priority H(v1, v2)}.

  while pqueue is not empty:
    v := pqueue.dequeue().
    mark v as visited.
    for each unvisited neighbor n of v:
      cost := v's cost +
              weight of edge (v, n).
      if cost < n's cost:
        n's cost := cost.
        n's previous := v.
        enqueue n at priority (cost + H(n, v2)).
  reconstruct path back from v2 to v1.
```

The pseudocode for Kruskal's is as follows:

kruskal(graph):

1. Place each vertex into its own "cluster" (group of reachable vertices).
2. Put all edges into a priority queue, using weights as priorities.
3. While there are two or more separate clusters remaining:
   - Dequeue an edge e from the priority queue.
   - If the start and finish vertices of e are not in the same cluster:
     - Merge the clusters containing the start and finish vertices of e.
     - Add e to your spanning tree.
   - Else:
     - Do not add e to your spanning tree.
4. Once the while loop terminates, your spanning tree is complete.