Dijkstra’s Algorithm and A*

C.S. Marron
cmarron@umbc.edu

CMSC 341 — Data Structures
Dijkstra’s Algorithm and A*

- The following slides on Dijkstra’s Algorithm and A* are from Programming Abstractions (CS 106X) at Stanford.
- Dijkstra’s Algorithm is covered in section 13.5 of your textbook.
Search

• Search **without** weights: What is the shortest path from A to D?
• Search with weights: What is the shortest path from A to D?
• **Search** **with** weights: What is the shortest path from A to D?
Least-Cost Paths

• BFS uses a **queue** to keep track of which nodes to use next

• BFS pseudocode:
  ```
  bfs from \( v_1 \):
  
  add \( v_1 \) to the queue.

  while queue is not empty:
    dequeue a node \( n \)
    enqueue \( n \)'s unseen neighbors
  ```

• How could we modify this pseudocode to dequeue the **least-cost** nodes instead of the **closest nodes**?
  – Use a **priority queue** instead of a queue
Edsger Dijkstra (1930-2002)

• famous Dutch computer scientist and prof. at UT Austin
  – Turing Award winner (1972)

• Noteworthy algorithms and software:
  – THE multiprogramming system (OS)
    • layers of abstraction
  – Compiler for a language that can do recursion
  – Dijkstra's algorithm
  – Dining Philosophers Problem: resource contention, deadlock
    • semaphores

• famous papers:
  – "Go To considered harmful"
  – "On the cruelty of really teaching computer science"
Dijkstra's Algorithm (18.6)

• **Dijkstra's algorithm**: Finds the minimum-weight path between a pair of vertices in a weighted directed graph.
  – Solves the "one vertex, shortest path" problem in weighted graphs.

  – *basic algorithm concept*: Create a table of information about the currently known best way to reach each vertex (cost, previous vertex), and improve it until it reaches the best solution.

• **Example**: In a graph where vertices are cities and weighted edges are roads between cities, Dijkstra's algorithm can be used to find the shortest route from one city to any other.
**Dijkstra pseudocode**

\[ \text{dijkstra}(v_1, v_2): \]

- consider every vertex to have a cost of infinity, except \( v_1 \) which has a cost of 0.
- create a *priority queue* of vertexes, ordered by cost, storing only \( v_1 \).

while the *pqueue* is not empty:

  - dequeue a vertex \( v \) from the *pqueue*, and mark it as **visited**.
  - for each of the unvisited neighbors \( n \) of \( v \), we now know that we can reach this neighbor with a total **cost** of (\( v \)'s cost + the weight of the edge from \( v \) to \( n \)).
    - if the neighbor is not in the *pqueue*, or this is cheaper than \( n \)'s current cost, we should **enqueue** the neighbor \( n \) to the *pqueue* with this new cost, and with \( v \) as its previous vertex.

when we are done, we can **reconstruct the path** from \( v_2 \) back to \( v_1 \) by following the previous pointers.
Dijkstra example

dijkstra(A, F);

- color key
  - white: unexamined
  - yellow: enqueued
  - green: visited

$v_1$'s distance := 0.
all other distances := $\infty$.

pqueue = \{A:0\}
Dijkstra example

dijkstra(A, F);

pqueue = \{D:1, B:2\}
Dijkstra example

dijkstra(A, F);

pqueue = \{B: 2, C: 3, E: 3, G: 5, F: 9\}
dijkstra(A, F);

pq = {C:3, E:3, G:5, F:9}
dijkstra(A, F);

pqueue = {E: 3, G: 5, F: 8}
Dijkstra example

dijkstra(A, F);

pqueue = {G:5, F:8}
Dijkstra example

dijkstra(A, F);

pqueue = \{F:6\}
Dijkstra example

dijkstra(A, F);

pq.queue = {}
dijkstra(A, F);
Algorithm properties

• Dijkstra's algorithm is a greedy algorithm:
  – Make choices that currently seem the best.
  – Locally optimal does not always mean globally optimal.

• It is correct because it maintains the following two properties:
  – 1) for every marked vertex, the current recorded cost is the lowest cost to that vertex from the source vertex.
  – 2) for every unmarked vertex $v$, its recorded distance is shortest path distance to $v$ from source vertex, considering only currently known vertices and $v$. 
Dijkstra exercise

• Run Dijkstra's algorithm from vertex E to all of the other vertices in the graph. (see next slide)
  – Keep track of previous vertices so that you can reconstruct the path.

  – Q: What path does it find from E to H?
    A. {E, B, F, C, H}
    B. {E, B, A, D, C, H}
    C. {E, G, A, B, F, D, C, H}
    D. {E, G, D, C, H}
    E. none of the above
Dijkstra tracing
Improving on Dijkstra’s

• If we want to travel from Stanford to San Francisco, Dijkstra's algorithm will look at path distances around Stanford. But, we know something about how to get to San Francisco -- we know that we generally need to go Northwest from Stanford.

• This is more information! Let's not only prioritize by weights, but also give some priority to the direction we want to go. E.g., we will add more information based on a heuristic, which could be direction in the case of a street map.
Finding a maze path

• Suppose we are searching for a path in a maze.
  – The 'cost' of a square is the min. number of steps we take to get there.
  – What does Dijkstra's algorithm do here? What "should" it do?
Dijkstra observations

- Dijkstra's algorithm uses a priority queue and examines possible paths in increasing order of their known cost or distance.
  - The idea is that paths with a lower distance-so-far are more likely to lead to paths with a lower total distance at the end.
  - But what about the remaining distance? What if we knew that a path that was promising so far will be unlikely to lead to a good result?
  - Can we modify the algorithm to take advantage of this information?
Dijkstra's algorithm works by incrementally computing the shortest path to intermediary nodes in the graph in case they prove to be useful.

– Some of these paths are in the "wrong" direction.

The algorithm has no "big-picture" conception of how to get to the destination; the algorithm explores outward in all directions.

– Could we give the algorithm a hint? Explore in a smarter order?
– What if we knew more about the vertices or graph being searched?
Heuristics

• **heuristic**: A speculation, estimation, or educated guess that guides the search for a solution to a problem.
  – *Example*: Spam filters flag a message as probable spam if it contains certain words, has certain attachments, is sent to many people, ...
  
  – In the context of graph searches: A function that approximates the distance from a known vertex to another destination vertex.
  – *Example*: Estimate the distance between two places on a Google Maps graph to be the direct straight-line distance between them.

• **admissible heuristic**: One that never overestimates the distance.
  – Okay if the heuristic underestimates sometimes (e.g. Google Maps).
  – Only ignore paths that in the *best case* are worse than your current path.
The A* algorithm

**A* ("A star"):** A modified version of Dijkstra's algorithm that uses a heuristic function to guide its order of path exploration.

Suppose we are looking for paths from start vertex $a$ to $c$.
- Any intermediate vertex $b$ has two costs:
  - The known (exact) cost from the start vertex $a$ to $b$.
  - The heuristic (estimated) cost from $b$ to the end vertex $c$.

**Idea:** Run Dijkstra's algorithm, but use this priority in the pqueue:
- $\text{priority}(b) = \text{cost}(a, b) + \text{Heuristic}(b, c)$
- Chooses to explore paths with lower estimated cost.
Example: Maze heuristic

• Suppose we are searching paths in a maze.
  – The 'cost' of a square is the min. number of steps we take to get there.
  – What would be a good heuristic for the remaining distance?
Maze heuristic

- Idea: Use "Manhattan distance" (straight-line) between the points.
  - $H(p_1, p_2) = \text{abs}(p_1.x - p_2.x) + \text{abs}(p_1.y - p_2.y)$ // dx + dy
  - The idea: Dequeue/explore neighbors with lower (cost+Heuristic).
Dijkstra

<table>
<thead>
<tr>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9?</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>9?</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>9?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>9?</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9?</td>
</tr>
</tbody>
</table>

The table represents a grid with numbers, possibly showing the result of Dijkstra's algorithm.
A*
Recall: Dijkstra code

dijkstra(v_1, v_2):
  consider every vertex to have a cost of infinity, except v_1 which has a cost of 0.
  create a priority queue of vertexes, ordered by cost, storing only v_1.

  while the pqueue is not empty:
    dequeue a vertex v from the pqueue, and mark it as visited.
    for each of the unvisited neighbors n of v, we now know that we can reach
    this neighbor with a total cost of (v's cost + the weight of the edge from v to n).
    if the neighbor is not in the pqueue, or this is cheaper than n's current cost,
    we should enqueue the neighbor n to the pqueue with this new cost,
    and with v as its previous vertex.

  when we are done, we can reconstruct the path from v_2 back to v_1
  by following the previous pointers.
**A* pseudocode**

\[ \text{astar}(v_1, v_2) : \]
consider every vertex to have a cost of infinity, except \( v_1 \) which has a cost of 0.
create a *priority queue* of vertexes, ordered by \((\text{cost} + \text{heuristic})\), storing only \( v_1 \) with a priority of \( H(v_1, v_2) \).

while the \( \text{pqueue} \) is not empty:
  dequeue a vertex \( v \) from the \( \text{pqueue} \), and mark it as \text{visited}.
  for each of the unvisited neighbors \( n \) of \( v \), we now know that we can reach this neighbor with a total \text{cost} of \((v's \text{ cost} + \text{ the weight of the edge from v to n})\).
    if the neighbor is not in the \( \text{pqueue} \), or this is cheaper than \( n \)'s current cost, we should \text{enqueue} the neighbor \( n \) to the \( \text{pqueue} \) with this new cost \text{plus} \( H(n, v_2) \), and with \( v \) as its previous vertex.

when we are done, we can \text{reconstruct the path} from \( v_2 \) back to \( v_1 \) by following the previous pointers.

* (basically, add \( H(...) \) to costs of elements in \( \text{PQ} \) to improve \( \text{PQ} \) processing order)