CSCI 104
Dijkstra’s algorithm and A*

Mark Redekopp
David Kempe
Sandra Batista
Dijkstra's Algorithm

SINGLE-SOURCE SHORTEST PATH (SSSP)
SSSP

- Assign to each edge a positive weight
  - Could be physical distance, cost of using the link, etc.
- Find the shortest path from a source node, 'a' to all other nodes

<table>
<thead>
<tr>
<th>List of Vertices</th>
<th>Adjacency Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(c,13),(e,4)</td>
</tr>
<tr>
<td>b</td>
<td>(c,5),(h,6)</td>
</tr>
<tr>
<td>c</td>
<td>(a,13),(b,5),(d,2),(e,8),(g,7)</td>
</tr>
<tr>
<td>d</td>
<td>(c,2),(f,1)</td>
</tr>
<tr>
<td>e</td>
<td>(a,4),(c,8),(f,3)</td>
</tr>
<tr>
<td>f</td>
<td>(d,1),(e,3),(g,4)</td>
</tr>
<tr>
<td>g</td>
<td>(c,7),(f,4),(h,14)</td>
</tr>
<tr>
<td>h</td>
<td>(b,6),(g,14)</td>
</tr>
</tbody>
</table>
SSSP

• What is the shortest distance from 'a' to all other vertices?

• **Distance** is defined to be **sum of weights on path between source and node**.

• How would you compute these distances?

<table>
<thead>
<tr>
<th>List of Vertices</th>
<th>Adjacency Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(c,13),(e,4)</td>
</tr>
<tr>
<td>b</td>
<td>(c,5),(h,6)</td>
</tr>
<tr>
<td>c</td>
<td>(a,13),(b,5),(d,2),(e,8),(g,7)</td>
</tr>
<tr>
<td>d</td>
<td>(c,2),(f,1)</td>
</tr>
<tr>
<td>e</td>
<td>(a,4),(c,8),(f,3)</td>
</tr>
<tr>
<td>f</td>
<td>(d,1),(e,3),(g,4)</td>
</tr>
<tr>
<td>g</td>
<td>(c,7),(f,4),(h,14)</td>
</tr>
<tr>
<td>h</td>
<td>(b,6),(g,14)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vert</th>
<th>Dist</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
</tr>
</tbody>
</table>
Dijkstra's Algorithm

• On each iteration, Dijkstra's algorithm selects vertex with minimum distance to source vertex.

• BFS uses *queue* to maintain vertices in order discovered.

• Dijkstra’s uses a *priority queue* to maintain vertices in shortest distance to source.
  
  – To demonstrate, we’ll use table of all vertices with their current known distances to source.

<table>
<thead>
<tr>
<th>Vert</th>
<th>Dist</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>inf</td>
</tr>
<tr>
<td>c</td>
<td>inf</td>
</tr>
<tr>
<td>d</td>
<td>inf</td>
</tr>
<tr>
<td>e</td>
<td>inf</td>
</tr>
<tr>
<td>f</td>
<td>inf</td>
</tr>
<tr>
<td>g</td>
<td>inf</td>
</tr>
<tr>
<td>h</td>
<td>inf</td>
</tr>
</tbody>
</table>
Dijkstra's Algorithm

1. SSSP(G, s)
2. PQ = empty PQ
3. s.dist = 0; s.pred = NULL
4. PQ.insert(s)
5. For all v in vertices
6. if v != s then v.dist = inf; PQ.insert(v)
7. while PQ is not empty
8. v = min(); PQ.remove_min()
9. for u in neighbors(v)
10. w = weight(v,u)
11. if(v.dist + w < u.dist)
12. u.pred = v
13. u.dist = v.dist + w;
14. PQ.decreaseKey(u, u.dist)
Dijkstra's Algorithm

1. SSSP(G, s)
2. PQ = empty PQ
3. s.dist = 0; s.pred = NULL
4. PQ.insert(s)
5. For all v in vertices
   6. if v != s then v.dist = inf; PQ.insert(v)
7. while PQ is not empty
   8. v = min(); PQ.remove_min()
9. for u in neighbors(v)
   10. w = weight(v,u)
   11. if(v.dist + w < u.dist)
   12. u.pred = v
   13. u.dist = v.dist + w;
   14. PQ.decreaseKey(u, u.dist)
Dijkstra's Algorithm

1. \text{SSSP}(G, s)
2. \text{PQ} = \text{empty PQ}
3. \text{s.dist} = 0; \text{s.pred} = \text{NULL}
4. \text{PQ.insert(s)}
5. \text{For all v in vertices}
6. \quad \text{if v} \neq \text{s then v.dist} = \text{inf; PQ.insert(v)}
7. \quad \text{while PQ is not empty}
8. \quad \text{v = min(); PQ.remove_min()}
9. \quad \text{for u in neighbors(v)}
10. \quad \quad \text{w = weight(v,u)}
11. \quad \quad \text{if(v.dist + w < u.dist)}
12. \quad \quad \text{u.pred} = \text{v}
13. \quad \text{u.dist} = \text{v.dist + w;}
14. \quad \text{PQ.decreaseKey(u, u.dist)
Dijkstra's Algorithm

1. SSSP(G, s)
2. PQ = empty PQ
3. s.dist = 0; s.pred = NULL
4. PQ.insert(s)
5. For all v in vertices
6. if v != s then v.dist = inf; PQ.insert(v)
7. while PQ is not empty
8. v = min(); PQ.remove_min()
9. for u in neighbors(v)
10. w = weight(v,u)
11. if(v.dist + w < u.dist)
12. u.pred = v
13. u.dist = v.dist + w;
14. PQ.decreaseKey(u, u.dist)
Dijkstra's Algorithm

1. SSSP(G, s)
2. PQ = empty PQ
3. s.dist = 0; s.pred = NULL
4. PQ.insert(s)
5. For all v in vertices
6. if v != s then v.dist = inf; PQ.insert(v)
7. while PQ is not empty
8. v = min(); PQ.remove_min()
9. for u in neighbors(v)
10. w = weight(v,u)
11. if(v.dist + w < u.dist)
12. u.pred = v
13. u.dist = v.dist + w;
14. PQ.decreaseKey(u, u.dist)
Dijkstra's Algorithm

1. SSSP(G, s)
2. PQ = empty PQ
3. s.dist = 0; s.pred = NULL
4. PQ.insert(s)
5. For all v in vertices
6. if v != s then v.dist = inf; PQ.insert(v)
7. while PQ is not empty
8. v = min(); PQ.remove_min()
9. for u in neighbors(v)
10. w = weight(v,u)
11. if(v.dist + w < u.dist)
12. u.pred = v
13. u.dist = v.dist + w;
14. PQ.decreaseKey(u, u.dist)
Dijkstra's Algorithm

1. SSSP(G, s)
2. PQ = empty PQ
3. s.dist = 0; s.pred = NULL
4. PQ.insert(s)
5. For all v in vertices
6. if v != s then v.dist = inf; PQ.insert(v)
7. while PQ is not empty
8. v = min(); PQ.remove_min()
9. for u in neighbors(v)
10. w = weight(v,u)
11. if(v.dist + w < u.dist)
12. u.pred = v
13. u.dist = v.dist + w;
14. PQ.decreaseKey(u, u.dist)

List of Vertices

<table>
<thead>
<tr>
<th>Vert</th>
<th>Dist</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>15</td>
</tr>
<tr>
<td>c</td>
<td>10</td>
</tr>
<tr>
<td>d</td>
<td>8</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
</tr>
<tr>
<td>f</td>
<td>7</td>
</tr>
<tr>
<td>g</td>
<td>11</td>
</tr>
<tr>
<td>h</td>
<td>inf</td>
</tr>
</tbody>
</table>
Dijkstra's Algorithm

1. SSSP(G, s)
2. PQ = empty PQ
3. s.dist = 0; s.pred = NULL
4. PQ.insert(s)
5. For all v in vertices
6. if v != s then v.dist = inf; PQ.insert(v)
7. while PQ is not empty
8. v = min(); PQ.remove_min()
9. for u in neighbors(v)
10. w = weight(v,u)
11. if(v.dist + w < u.dist)
12. u.pred = v
13. u.dist = v.dist + w;
14. PQ.decreaseKey(u, u.dist)
Dijkstra's Algorithm

1. SSSP(G, s)
2. PQ = empty PQ
3. s.dist = 0; s.pred = NULL
4. PQ.insert(s)
5. For all v in vertices
   6. if v != s then v.dist = inf; PQ.insert(v)
7. while PQ is not empty
   8. v = min(); PQ.remove_min()
9. for u in neighbors(v)
   10. w = weight(v,u)
   11. if(v.dist + w < u.dist)
   12. u.pred = v
   13. u.dist = v.dist + w;
   14. PQ.decreaseKey(u, u.dist)
Analysis

- What is the loop invariant?
  - The vertex v removed from PQ is guaranteed to be the vertex with shortest path to source of all vertices in PQ and its distance is set to its shortest distance to the source.
  - All vertices whose distances and predecessors are set have shortest known distance thus far to source.

- Proof sketch by induction
  - First node from PQ is source itself with distance 0 to itself.
  - Decrease the distance to its neighbors; its neighbor with the shortest distance will be at front of PQ
  - No shorter path from source to vertex at front of PQ; any other path would use some edge from the start having greater distance
A* Search Algorithm

ALGORITHM HIGHLIGHT
Search Methods

• Many systems require searching for goal states
  – Path Planning
    • Mapquest/Google Maps
    • Games
  – Optimization Problems
    • Find the optimal solution to a problem with many constraints
Search Applied to 8-Tile Game

• 8-Tile Puzzle
  – 3x3 grid with one blank space
  – With a series of moves, get the tiles in sequential order
  – Goal state:

![8-Tile Puzzle Diagram]

Original: No order

Goal State
Search Methods

• **Brute-Force Search**: Search all possibilities until you find it! 😞

• **Heuristic Search**: A heuristic is a “rule of thumb”.
  – Heuristics are not perfect; they are quick computations to give an approximate measure.
Brute Force Search

- Brute Force Search Tree
  - Generate all possible moves
  - Explore each move despite its proximity to the goal node
Heuristics

• Heuristics are “scores” of how close a state is to the goal (usually, lower = better)
• Heuristics must be easy to compute from current state
• Heuristics for 8-tile puzzle
  – # of tiles out of place
  – Total x-, y- distance of each tile from its correct location (Manhattan distance)
Heuristic Search

- Heuristic Search Tree
  - Use total $x$-$y$-distance (Manhattan distance) heuristic
  - Explore the lowest scored states
Caution About Heuristics

• Heuristics may be wrong
• Sometimes pursuing lowest heuristic score leads to a less-than optimal solution or even no solution
• Solution
  – Take # of moves from start (depth) into account
A-star Algorithm

• Use a new metric to decide which state to explore/expand

• Define
  – \( h = \) heuristic score (Manhattan distance)
  – \( g = \) number of moves from start to current state
  – \( f = g + h \)

• As we explore states and their successors, assign each state its \( f \)-score and always explore the state with lowest \( f \)-score

• Heuristics should always underestimate the distance to the goal
  – If so, A* guarantees optimal solutions
A-Star Algorithm

• Maintain 2 lists
  – Open list = Nodes to be explored (chosen from)
  – Closed list = Nodes already explored (already chosen)

• Pseudocode

```python
open_list.push(Start State)
while(open_list is not empty)
    1. s ← remove min. f-value state from open_list
       (if tie in f-values, select one w/ larger g-value)
    2. Add s to closed list
    3a. if s = goal node then trace path back to start; STOP!
    3b. For each neighbor, v, of s, compute f-value
        if v on closed_list, skip v.
        if v not on open_list, add v to open list
        if v on open_list, update f-value if current value is smaller
```
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list
     (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start; STOP!
  3b. For each neighbor, v, of s, compute f-value
      if v on closed_list, skip v.
      if v not on open_list, add v to open list
      if v on open_list, update f-value if current value is smaller
**Path-Planning w/ A* Algorithm**

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

```python
open_list.push(Start State)
while(open_list is not empty):
  1. s ← remove min. f-value state from open_list
     (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
     3a. if s = goal node then trace path back to start; STOP!
     3b. For each neighbor, v, of s, compute f-value
         if v on closed_list, skip v.
         if v not on open_list, add v to open list
         if v on open_list, update f-value if current
          value is smaller
```
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list
     (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start; STOP!
  3b. For each neighbor, v, of s, compute f-value
      if v on closed_list, skip v.
      if v not on open_list, add v to open_list
      if v on open_list, update f-value if current value is smaller
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list
     (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start; STOP!
  3b. For each neighbor, v, of s, compute f-value
     if v on closed_list, skip v.
     if v not on open_list, add v to open list
     if v on open_list, update f-value if current value is smaller

```cpp
S

S

G
```

```cpp
S

S

G
```
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list
     (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start; STOP!
  3b. For each neighbor, v, of s, compute f-value
      if v on closed_list, skip v.
      if v not on open_list, add v to open list
      if v on open_list, update f-value if current value is smaller
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. \( s \leftarrow \) remove min. \( f \)-value state from open_list
     (if tie in \( f \)-values, select one w/ larger \( g \)-value)
  2. Add \( s \) to closed list
  3a. if \( s \) = goal node then trace path back to start; STOP!
  3b. For each neighbor, \( v \), of \( s \), compute \( f \)-value
      if \( v \) on closed_list, skip \( v \).
      if \( v \) not on open_list, add \( v \) to open list
      if \( v \) on open_list, update \( f \)-value if current value is smaller
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list
     (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start;
      STOP!
  3b. For each neighbor, v, of s, compute f-value
      if v on closed_list, skip v.
      if v not on open_list, add v to open list
      if v on open_list, update f-value if current value is smaller
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list
     (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start;
      STOP!
  3b. For each neighbor, v, of s, compute f-value
      if v on closed_list, skip v.
      if v not on open_list, add v to open list
      if v on open_list, update f-value if current
         value is smaller
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list
     (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start;
      STOP!
  3b. For each neighbor, v, of s, compute f-value
     if v on closed_list, skip v.
     if v not on open_list, add v to open list
     if v on open_list, update f-value if current
        value is smaller
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

```
open_list.push(Start State)
while(open_list is not empty)
    1. s ← remove min. f-value state from open_list
       (if tie in f-values, select one w/ larger g-value)
    2. Add s to closed list
    3a. if s = goal node then trace path back to start;
       STOP!
    3b. For each neighbor, v, of s, compute f-value
        if v on closed_list, skip v.
        if v not on open_list, add v to open list
        if v on open_list, update f-value if current
           value is smaller
```
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan (x-/y-) distance

```
open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list
     (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start;
      STOP!
  3b. For each neighbor, v, of s, compute f-value
      if v on closed_list, skip v.
      if v not on open_list, add v to open list
      if v on open_list, update f-value if current
         value is smaller
```
Path-Planning w/ A* Algorithm

• Find optimal path from S to G using A*
  – Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list
     (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start; STOP!
  3b. For each neighbor, v, of s, compute f-value
      if v on closed_list, skip v.
      if v not on open_list, add v to open list
      if v on open_list, update f-value if current value is smaller
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. $f$-value state from open_list
     (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start; STOP!
  3b. For each neighbor, v, of s, compute $f$-value
      if v on closed_list, skip v.
      if v not on open_list, add v to open list
      if v on open_list, update $f$-value if current value is smaller
Path-Planning w/ A* Algorithm

- Find optimal path from S to G using A*
  - Use heuristic of Manhattan (x-/y-) distance

```python
open_list.push(Start State)
while(open_list is not empty)
  1. s ← remove min. f-value state from open_list
     (if tie in f-values, select one w/ larger g-value)
  2. Add s to closed list
  3a. if s = goal node then trace path back to start;
      STOP!
  3b. For each neighbor, v, of s, compute f-value
      if v on closed_list, skip v.
      if v not on open_list, add v to open list
      if v on open_list, update f-value if current
      value is smaller
```
A* and BFS

• BFS explores all nodes at a shorter distance from the start (i.e. g value)
A* and BFS

- BFS explores all nodes at a shorter distance from the start (i.e. g value)
A* and BFS

- BFS is A* using just the g value to choose which item to select and expand.
A* Analysis

- What data structure should we use for the open-list?
- What data structure should we use for the closed-list?
- A* is essentially modification of Dijkstra’s with heuristic used for priorities in PQ
- Run time is similar to Dijkstra's algorithm...
  - Each node added/removed once from the open-list so that incurs N*O(remove-cost)
  - Visiting each neighbor requires O(E) operations and performing an insert or decrease operation is E*max(O(insert), O(decrease))
  - E = Number of potential successors and this depends on the problem and the possible solution space
  - For the tile puzzle game, how many potential boards are there? (This is why there was a sad face next to the brute force approach...)