Part III

Uninformed versus heuristic search

Uniform-cost search from Canterbury to Harrietsham

0. [(0.0, [cant])]
1. [(5.63, [cant, st]), (6.92, [cant, chart]), (11.59, [cant, whit]), (31.38, [cant, bar]), (28.97, [cant, sand]), (21.40, [cant, fav])]
2. [(6.92, [cant, chart]), (21.40, [cant, fav]), (11.59, [cant, whit]), (31.38, [cant, bar]), (28.97, [cant, sand]), (15.13, [cant, st, hb]), (36.69, [cant, st, mar]), (39.43, [cant, st, rams])]
3. [(11.59, [cant, whit]), (21.40, [cant, fav]), (15.13, [cant, st, hb]), (26.71, [cant, chart, ash]), (28.97, [cant, sand]), (39.43, [cant, st, rams]), (36.69, [cant, st, mar]), (31.38, [cant, bar]), (46.83, [cant, chart, harr])]
4. [(15.13, [cant, st, hb]), (21.40, [cant, fav]), (36.69, [cant, st, mar]), (26.71, [cant, chart, ash]), (24.78, [cant, whit, hb]), (39.43, [cant, st, rams]), (46.83, [cant, chart, harr]), (31.38, [cant, bar]), (27.20, [cant, whit, fav]), (28.97, [cant, sand])]

…

Uniform-cost search from Canterbury to Harrietsham (cont’)

16. [(44.58, [cant, st, mar, rams]), (47.47, [cant, bar, folk]), (46.83, [cant, chart, harr]), (50.21, [cant, chart, ash, tent]), (50.21, [cant, fav, whit, hb]), (47.31, [cant, st, rams, mar]), (52.46, [cant, chart, ash, harr]), (52.14, [cant, chart, ash, hy]), (51.18, [cant, st, rams, sand]), (50.37, [cant, bar, dov]), (65.34, [cant, whit, hb, st, mar]), (48.44, [cant, chart, ash, fav]), (48.92, [cant, whit, fav, ash]), (60.51, [cant, chart, ash, folk]), (65.66, [cant, st, hb, whit, fav, ash]), (52.46, [cant, st, hb, mar, rams]), (52.30, [cant, chart, ash, nr]), (57.29, [cant, chart, ash, rye]), (68.07, [cant, whit, hb, st, rams]), (52.95, [cant, sand, dov]), (54.23, [cant, whit, hb, mar]), (74.51, [cant, sand, rams, st]), (66.14, [cant, st, mar, hb]), (66.63, [cant, fav, ash, tent]), (48.60, [cant, sand, rams, mar]), (62.92, [cant, fav, ash, chart]), (76.92, [cant, fav, ash, folk]), (68.88, [cant, fav, ash, harr]), (68.56, [cant, fav, ash, hy]), (68.72, [cant, fav, ash, nr]), (73.71, [cant, fav, ash, rye]), (58.26, [cant, sand, deal, dov])]
17. [(46.83, [cant, chart, harr]), …, (56.33, [cant, st, mar, rams, sand])]

Best-first search from Canterbury to Harrietsham

0. [(45.68, [cant])]

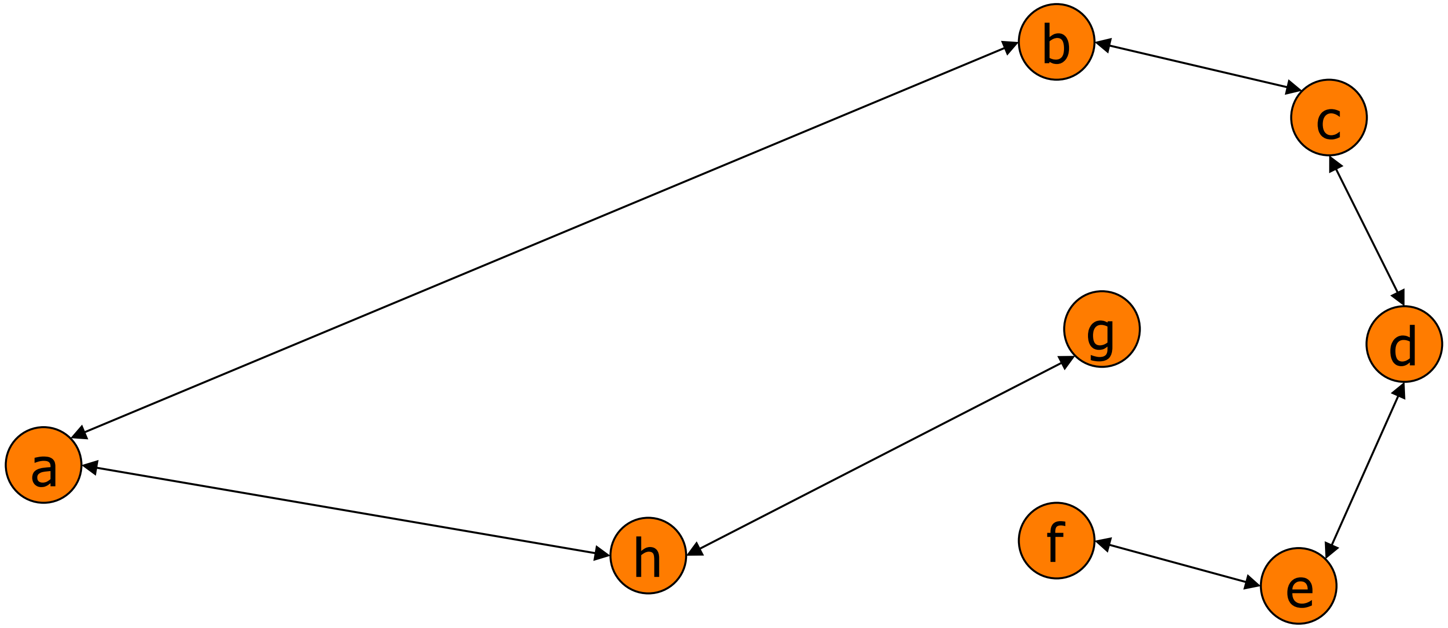
1. [(25.99, [cant, fav]), (50.78, [cant, st]), (39.21, [cant, chart]), (74.39, [cant, sand]), (54.65, [cant, bar]), (41.51, [cant, whit])]

2. [(25.10, [cant, fav, ash]) , (50.78, [cant, st]), (39.21, [cant, chart]), (74.39, [cant, sand]), (54.65, [cant, bar]), (41.51, [cant, whit]), (41.51, [cant, fav, whit])]

3. [(0.00, [cant, fav, ash, harr]), (39.21, [cant, chart]), (19.52, [cant, fav, ash, tent]), (50.78, [cant, st]), (41.63, [cant, fav, ash, nr]), (33.55, [cant, fav, ash, rye]), (41.51, [cant, fav, whit]), (74.39, [cant, sand]), (58.19, [cant, fav, ash, folk]), (54.65, [cant, bar]), (49.63, [cant, fav, ash, hy]), (41.51, [cant, whit]), (39.21, [cant, fav, ash, chart])]

- Route has length of 68.88 kms, hence best-first is sub-optimal (uniform-cost route is 46.83 kms)

Backing out of false paths with best-first search



Backing out of false paths when travelling from a to g

0. [(8.5, [a])]
1. [(2.3, [a,b]), (4.1, [a,h])]
2. [(2.4, [a,b,c]), (4.1, [a,h])]
3. [(2.4, [a,b,c,d]), (4.1, [a,h])]
4. [(2.5, [a,b,c,d,e]), (4.1, [a,h])]
5. [(1.6, [a,b,c,d,e,f]), (4.1, [a,h])]
6. [(4.1, [a,h])]
7. [(0.0, [a,h,g])]

Optimality of uniform-cost

- When a route is expanded, ***all*** routes that are strictly smaller have already been expanded
- Suppose that r is the ***first*** route that is up for expansion that already leads to a goal state
- Route r is a solution but ***assume*** that it is not optimal
- Then a smaller route r′ must exist
- The route r′ would be expanded earlier than r
- Hence r would not be the first route for expansion which leads to a goal state -- a contradiction

Completeness of best-first (and related algorithms)

- Déjà vu check ensures that no town occurs multiply in a route
- Thus each town can occur at most once
- Consider the number of routes possible with just n = 3 towns a, b and c:
  - [a,b,c], [a,c,b], [b,a,c], [b,c,a], [c,a,b], [c,b,a] (3!)
  - [a,b], [b, a], [a, c], [c, a], [b, c], [c, b] (3!)
  - [a], [b], [c] (≤3!)
- Number of different routes is there (≤n(n!)) which is finite whenever n is finite
- Since no route is ever expanded twice, best-first will either:
  - Terminate by expanding all routes without finding a solution (case 2)
  - Terminate earlier by finding a solution (case 1)
- Therefore best-first search is complete

Pair class (1 of 2)

```
import java.util.*;

public class Pair implements Comparable<Pair>
{
    private double rank;
    private LinkedList<Town> route;

    public double getRank()
    {
        return rank;
    }

    public LinkedList<Town> getRoute()
    {
        return route;
    }
```

Pair class (2 of 2)

```
Pair(double rank, LinkedList<Town> route)
{
        this.rank = rank;
        this.route = route;
}

public int compareTo(Pair pair)
{
        if (rank > pair.getRank()) return 1;
        else if (rank < pair.getRank()) return -1;
        else return 0;                  // return (int) (rank - pair.getRank());
}

public String toString()                // force short debug traces
{
        return "(" + String.format("%.2f", rank) + ", " + route + ")";
}
}
```

Uniform-cost and best-first methods (1 of 2)

```
private LinkedList<Town> uniformCost(Town start, Town dest)
{
        LinkedList<Town> route = new LinkedList<Town>();
        route.add(start);
        PriorityQueue pairs = new PriorityQueue();
        pairs.add(new Pair(0.0, route)); // uniform-cost
//      pairs.add(new Pair(estimateDistance(start, dest), route)); // best
        while (true)
        {
//          System.out.println(pairs);              // debug traces
            if (pairs.size() == 0) return null;     // no solutions exist
            Pair pair = (Pair) pairs.poll();        // retrieve and remove (log)
            route = pair.getRoute();
            Town last = route.getLast();
            if (last.equals(dest)) return route;    // exit loop with solution
            LinkedList<Town> nextTowns = graph.get(last);
            for (Town next:nextTowns)
```

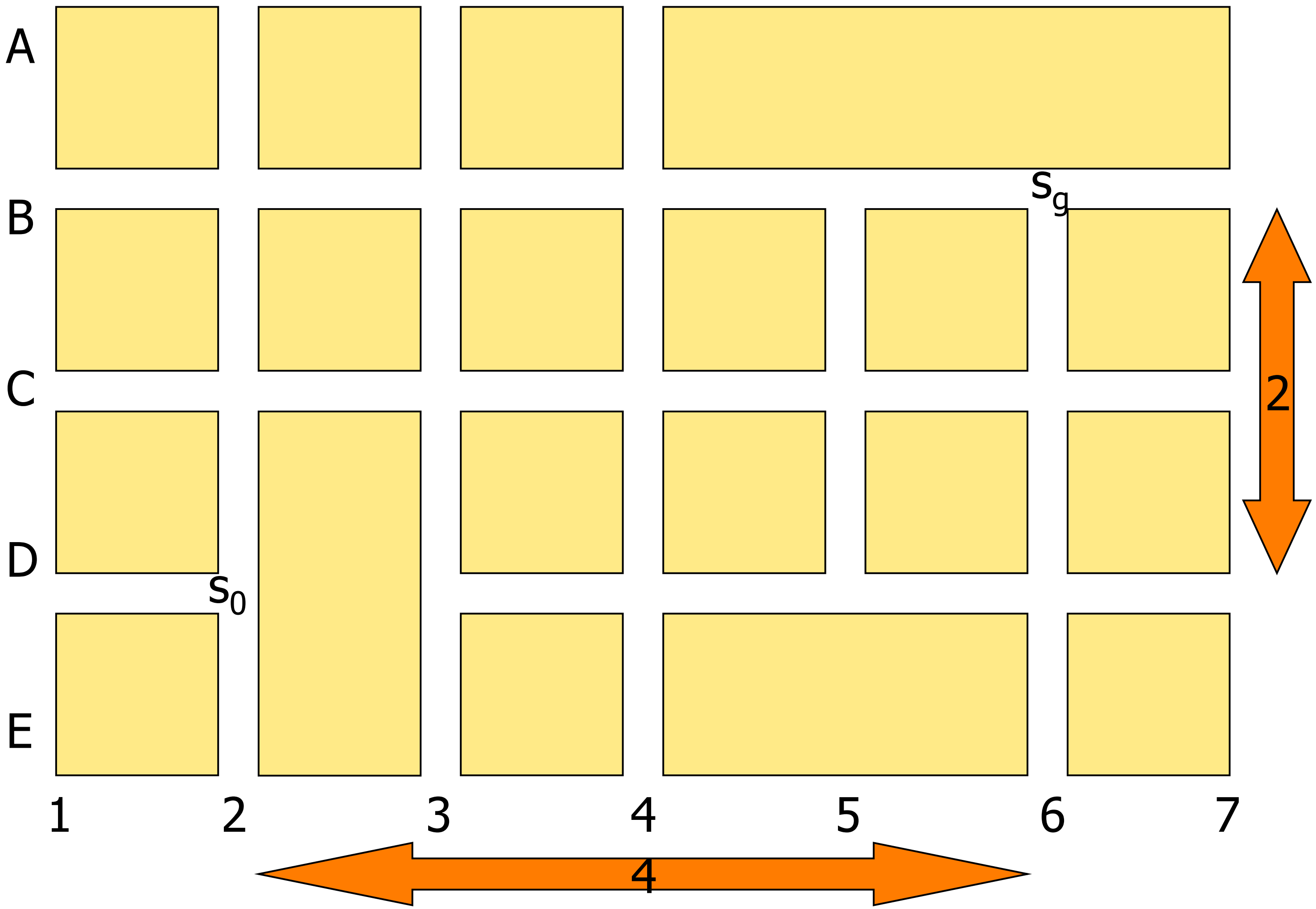
Uniform-cost and best-first methods (2 of 2)

```
        for (Town next:nextTowns)
        {
            if (!route.contains(next))                         // deja vu
            {
                LinkedList<Town> nextRoute = new LinkedList<Town>(route);
                nextRoute.addLast(next);
                double distance = actualDistance(nextRoute);       // uniform
//              double distance = estimateDistance(next, dest); // best-first
                pairs.add(new Pair(distance, nextRoute));          // log too
            }
        }
    }
}
```

Heuristics in AI

- Greeks: “heuriskekein” means to “to find” (so Archimedes shouted “Heureka”)
- 60s: heuristic as opposed to algorithmic: “a process that may solve a given problem, but offers no guarantee of doing so”, [Newell and Simon, *Lernende Automaten* (Automata), 1963]
- 70s: heuristic programming used for expert systems/rule-based programming in which “rules of thumb” were extracted from domain experts
- 80s: a process that improves average-case performance of an ***algorithm*** but does not necessarily improve worse-case performance

Straight-line distance = sqrt(20) but City block distance = 6



Heuristic for 8-puzzle (not needed for the assessment)

1	7	3
4	_	2
6	8	5

s_i → s_g

1	2	3
4	5	6
7	8	_

- Tiles 7, 2, 6 and 5 in s_i are out-of-place (the blank is not a tile)
- Each move can correct at most of one tile
- Thus an admissible heuristic is the total number of tiles that remain out-of-place (4 for s_i)

Better heuristic for 8-puzzle

1	7	3
4	_	2
6	8	5

s_i → s_g

1	2	3
4	5	6
7	8	_

- Each move can move at ***most of one tile one position nearer*** its destination
- Tile 7 requires 1 horizontal and 2 vertical moves
- Minimum of (1+2) + (1+1) + (2+1) + (1+1) = 3+2+3+2 = 10 moves required for s_i

Evaluation (ranking) functions

- Evaluation function $e(r, s_g)$ takes a route r and a goal state s_g and gives a number that represents the desirability of expanding r
- For uniform-cost search: $e(r, s_g) = g(r)$ where $g(r)$ represents the cost of r
- For best-first search: $e(r, s_g) = h(r, s_g)$ where h is a heuristic that estimates the cost of travelling on from last state in r to s_g
- For A* search: $e(r, s_g) = g(r) + h(r, s_g)$ [with a technical caveat on $h(r, s_g)$]

A* search from Canterbury to Harrietsham

$e(r, s_g) = g(r) + h(r, s_g)$ is an estimate of cost of a complete journey that progresses along the route r and then continues onto s_g

0. [(45.68, [cant])]

1. [(46.13, [cant, chart]) , (56.41, [cant, st]), (47.40, [cant, fav]), (103.36, [cant, sand]), (86.03, [cant, bar]), (53.10, [cant, whit])]

2. [(46.83, [cant, chart, harr]), (56.41, [cant, st]), (47.40, [cant, fav]), (103.36, [cant, sand]), (86.03, [cant, bar]), (53.10, [cant, whit]), (51.81, [cant, chart, ash])]

A* method (1 of 2)

```
private LinkedList<Town> aStar(Town town1, Town town2)
{
        LinkedList<Town> route = new LinkedList<Town>();
        route.add(town1);
        PriorityQueue pairs = new PriorityQueue();
        pairs.add(new Pair(estimateDistance(town1, town2), route)); // A*

        while (true)
        {
//              System.out.println(pairs);                  // debug traces
                if (pairs.size() == 0) return null;         // no solutions exist
                Pair pair = (Pair) pairs.poll();            // retrieve and remove (log)
                route = pair.getRoute();
                Town last = route.getLast();
                if (last.equals(town2)) return route;       // exit loop with solution
                LinkedList<Town> nextTowns = graph.get(last);
                for (Town next:nextTowns)
```

A* method (2 of 2)

```
    for (Town next:nextTowns)
    {
        if (!route.contains(next))                              // deja vu
        {
            LinkedList<Town> nextRoute = new LinkedList<Town>(route);
            nextRoute.addLast(next);
            double distance = actualDistance(nextRoute);       // A*
            distance += estimateDistance(next, town2);        // A*
            pairs.add(new Pair(distance, nextRoute));          // log too
        }
    }
  }
}
```

Expansion counts (efficiency)

search problem		*unif*	*best*	*A**	*length*
canterbury	gillingham	297	6	32	109.75
dover	tenterden	48	3	5	74.19
maidstone	dungeness	78627	78627	78627	none
deal	faversham	25	3	3	65.02
sheerness	cranbrook	6	3	3	65.66
sittingbourne	sandwich	72	7	7	118.12
ashford	ramsgate	56	4	4	66.14
new_romney	whitstable	23	4	5	62.92
barham	gravesend	464	6	16	121.34

Technical caveat revealed

- A* is a search algorithm that uses the evaluation function $e(r, s_g) = g(r) + h(r, s_g)$ where $h(r, s_g)$ is admissible
- A heuristic $h(r, s_g)$ is admissible if and only if:
  - it does ***not over-estimate*** the distance from the end of the route r to the goal state s_g
- Examples of admissible heuristics:
  - straight-line distance (using polar radius a for r)
  - Manhattan block distance
  - both 8-puzzle move heuristics

Optimality of A*

- Suppose the start state is s_0, s_g is the ***single*** goal state and $r = [s_0, s_1, \ldots, s_j, s_g]$ is ***an*** optimal route from s_0 to s_g
- ***Assume*** A* returns $r' = [s_0, s'_1, \ldots, s'_k, s_g]$ and ***$g(r) < g(r')$***
- Let $j = \max\{ n \mid [s_0, s_1, \ldots, s_n] = [s_0, s'_1, \ldots, s'_n]\}$
- Path $[s_0, s_1, \ldots, s_{j+1}]$ was never expanded by A* thus:
  - $g([s_0, s'_1, \ldots, s'_{j+1}]) + h(s'_{j+1}) \leq g([s_0, s_1, \ldots, s_{j+1}]) + h(s_{j+1})$
  - $g([s_0, s'_1, \ldots, s'_{j+1}, s'_{j+2}]) + h(s'_{j+2}) \leq g([s_0, s_1, \ldots, s_{j+1}]) + h(s_{j+1})$
  - …
  - $g([s_0, s'_1, \ldots, s'_{j+1}, s'_{j+2}, \ldots, s_g]) + h(s_g) \leq g([s_0, s_1, \ldots, s_{j+1}]) + h(s_{j+1})$
- Since h is ***admissible***, $g([s_0, s_1, \ldots, s_{j+1}]) + h(s_{j+1}) \leq g(r)$
- Therefore $g([s_0, s'_1, \ldots, s'_{j+1}, s'_{j+2}, \ldots, s_g]) + h(s_g) \leq g(r)$
- Thus $g(r') + h(s_g) \leq g(r)$ hence $g(r') \leq g(r)$ which is a ***contradiction***
- Thus $g(r) \geq g(r')$ and since r is optimal it follows ***$g(r)=g(r')$***

Summary statement

	optimal	*complete*	*efficient*
breadth-first	depends	yes	no
uniform-cost	yes	yes	no
best-first	no	yes	yes
A*	yes	yes	yes