5SENG001W - Algorithms, Week 10

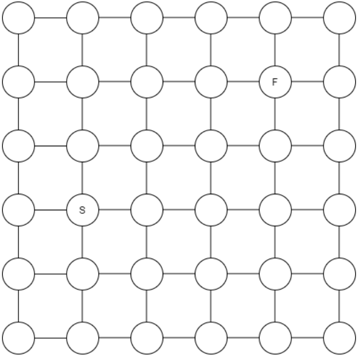
Dr. Klaus Draeger

Recap: Dijkstra's algorithm

- Dijkstra's algorithm does the following:
  - For each vertex v, keep track of the shortest known distance $d(s, v)$ from s to v
    - Many versions of the algorithm initially set all distances to ∞ except s, where it is 0
    - Our **closed list** represents those vertices where we have found a non-∞ distance
  - In each iteration, expand a vertex v
    - This means exploring its outgoing edges (v, x)
    - v must be a previously unexpanded vertex with minimal $d(s, v)$
    - Our **open list** contains the candidates for this
  - During the expansion, update $d(s, x)$ if $d(s, v) + w(v, x)$ is smaller

Towards A*

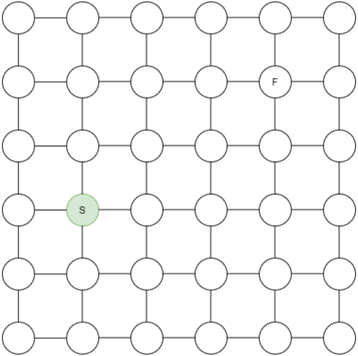
- Suppose we want to get from S to F in this graph:



- Why do we bother exploring to the south and west?

Towards A*

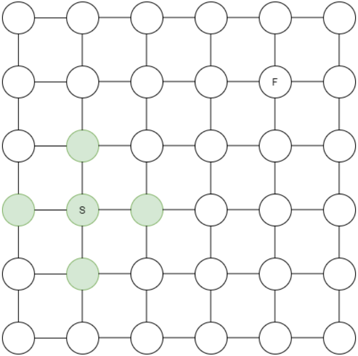
- Suppose we want to get from S to F in this graph:



- Why do we bother exploring to the south and west?

Towards A*

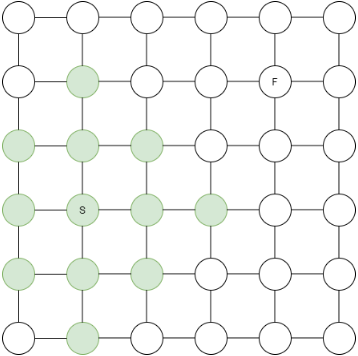
- Suppose we want to get from S to F in this graph:



- Why do we bother exploring to the south and west?

Towards A*

- Suppose we want to get from S to F in this graph:



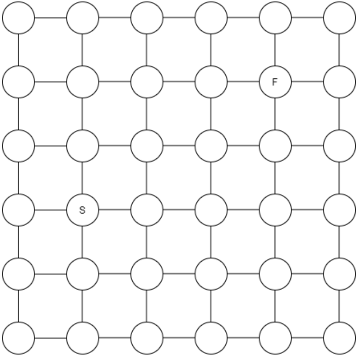
- Why do we bother exploring to the south and west?

Towards A*

- In the grid example, we know exactly where to go
  - We have a notion of directions, and know in which direction the target is
  - We know **how far** any given vertex is from the target
- So pathfinding becomes trivial
- In general, we don't have this exact information
  - We would need to already have solved the path-finding problem
- But we may have a suitable **estimate** of distances to the target

Towards A*

- For example, if the edges in the grid

  have unknown weights on them
  - The edges on the direct paths may have higher weights
  - So it may make sense to take a detour
  - But the distance in the unweighted version ("as the crow flies") can still be used to steer the search
  - We use it to bias the exploration to vertices that are "in the right direction"

The A* algorithm

- Like Dijkstra's algorithm, A* organises its open list as a priority queue
- The value associated with a vertex v is $f(v) + h(v)$, where
  - $f(v)$ is the distance from the start to v (discovered during the exploration)
  - $h(v)$ is an estimate for the distance from v to the target (must be provided)

Estimate functions

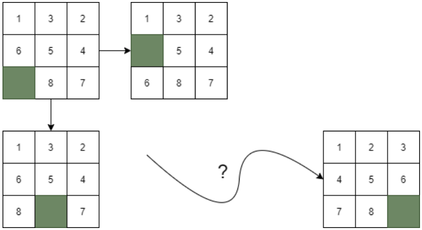
- A* relies on an estimate function h to work.
- This function should have some properties in order to work well:
  - It needs to be cheap to compute for each vertex
  - It needs to be an **underapproximation**, i.e. $h(v)$ cannot be greater than the actual distance to the target (otherwise a suboptimal path might be found first)
- How to find such an estimate?

State graphs and relaxations

- Common case: the graph represents some **system**
  - Vertices are **states** of the system
  - Edges are **transitions**
  - We want to go from some initial to some final state
- This graph is potentially huge
  - We do **not** want to create the whole graph before looking for a path
  - Instead, we want to create vertices only when needed ("on the fly")

State graphs: example

- We previously saw this tile sliding puzzle
  - Eight tiles numbered 1,. . . ,8 in a 3*3 square (bigger versions also exist)
  - One hole which adjacent tiles can be moved into
  - Goal is to have all the tiles in order



State graphs: example

- The state graph for the 9-puzzle has
  - One vertex for each permutation of the tiles and hole
  - Edges corresponding to moves in the puzzle
- A total of $9! = 362880$ vertices
  (and $16! = 20,922,789,888,000$ for the 4*4 version)
  so we really don't want to construct the whole graph
- On-the-fly search using A* helps avoid this

State graphs and relaxations

- Estimate functions for state graphs can be found based on **relaxations**
- Idea: relax the interaction between parts, so that
  - They can move independently
    (so it is easier to compute the distance in the relaxation)
  - All moves in the original system are still in the relaxation
    (so the distance in the relaxation is an underapproximation)
- Let's have a look at how this works in the 9-puzzle

State graphs: example

- In the 9-puzzle, the parts are the numbered tiles
- Their moves are not independent
  (because they cannot share a position)
- We can relax the system by removing this constraint
- In the relaxation, we can move each tile to its place independently
- The distance in the relaxation is the sum of each tile's distance to its target position

State graphs: example

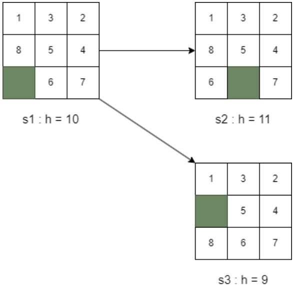
- For example, suppose we start in this state $s1$:

1	3	2
8	5	4
	6	7

  gets the distance estimate $h(s1) = 10$, because
  - Tiles 1 and 5 are in their target position
  - Tiles 2,3 are 1 step away from their target location
  - Tiles 4,6,7,8 are 2 steps away from their target location
  - $0 + 1 + 1 + 2 + 0 + 2 + 2 + 2 = 10$

State graphs: example

- The estimates for the successor states are 9 and 11:



- Then the search progresses like this:
  - The distances from the start are $f(s1) = 0, f(s2) = f(s3) = 1$.
  - The priorities are $f(s1) + h(s1) = 10, f(s2) + h(s2) = 12, f(s3) + h(s3) = 10$.
  - So $s3$ comes before $s2$ in the priority queue.