Graph Algorithms II

Bernd Kiefer
Jörg Steffen

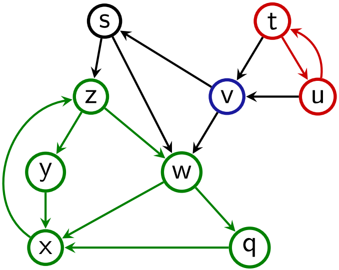
January 20, 2023

Heuristic Search Pictures from
`http://www-cs-students.stanford.edu/~amitp/gameprog.html`

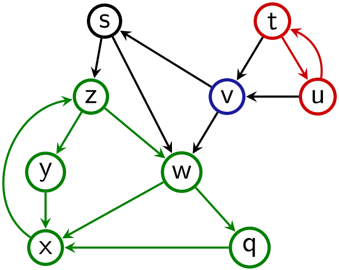
Strongly Connected Components

- Definition: a SCC of a directed graph is the maximal set U of vertices, such that for all $u, v \in U : u \rightarrow v \wedge v \rightarrow u$
- SCCs consist of connected cycles of the graph
- Vertices not in any cycle constitute their own SCC
- The SCCs form a total partition of the graph
- The component graph, where the SCCs are replaced by vertices, is acyclic
- Many algorithms are easier to solve on acyclic graphs
  - Run the algorithm on the harder, but smaller SCCs
  - Combine the results on the acyclic component graph
  - a special kind of divide and conquer

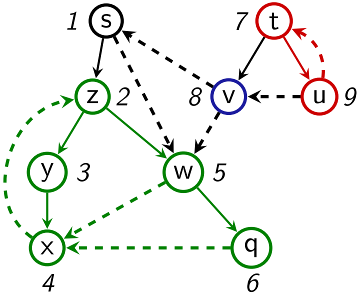
Strongly Connected Components



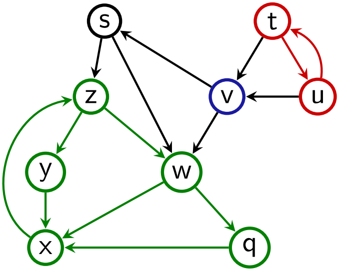
Strongly Connected Components



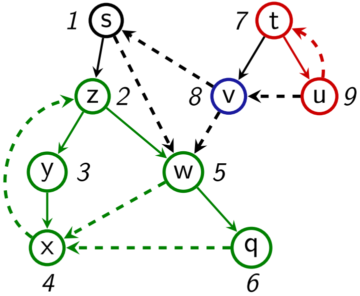
DFS starting at s then t



Strongly Connected Components



DFS starting at s then t



- All nodes of a SCC will be visited in the same DFS
- All vertices of an SCC are connected by tree edges
- There must be a highest entry node
- It is the vertex with the lowest discovery time in the SCC

Tarjan's Algorithm

- Crucial observation: Root Property
  - store, for every vertex, the lowest discovery time of an active vertex reachable from it in the DFS tree $\mapsto$ $low(v)$
  - a vertex with $d(v) == low(v)$ is the root of an SCC
  - the SCC consists of the root and all vertices that
    - are on the DFS tree below the root and
    - don't belong to another SCC

    $\longrightarrow$ we can use a stack to collect these vertices
  - $low(v) < d(v)$ can only occur when using
    - back edges
    - cross edges pointing to a vertex still on the stack (active SCC)
  - vertices not on the stack are not considered because they belong to an already finished SCC in another DFS tree branch
  - Pop the SCC vertices when reaching the root

Tarjan's SCC Algorithm

StronglyConnectedComponents(G)

Initialize all $v \in \mathcal{V}$: $d(v) = 0; low(v) = 0$;
for all $v \in \mathcal{V}$ with $d(v) == 0$ **do**
    $low(v) = d(v) = ++\,time$; S.push(v);
    for all $e = (v, u) \in \mathcal{E}$ **do**
        if $d(u) == 0$ **then**   // u not visited: recurse
            findSCC(u) ; $low(v) = min(low(v), low(u))$
        else if u is in S **then**
            $low(v) = min(low(v), low(u))$
    if $d(v) == low(v)$ **then**
        while S.top()$\neq v$ **do**
            $u = S$.pop();
        S.pop()   // pop root

Tarjan's SCC Algorithm

StronglyConnectedComponents(G)

Initialize all $v \in \mathcal{V}$: $d(v) = 0$; $low(v) = 0$;
for all $v \in \mathcal{V}$ with $d(v) == 0$ **do**
  $low(v) = d(v) = ++time$; S.push(v);
  for all $e = (v, u) \in \mathcal{E}$ **do**
    if $d(u) == 0$ **then** // u not visited: recurse
      findSCC(u) ; $low(v) = min(low(v), low(u))$
    else if u is in S **then**
      $low(v) = min(low(v), low(u))$
  if $d(v) == low(v)$ **then**
    while S.top()$\neq v$ **do**
      $u = S$.pop();
    S.pop() // pop root

To check efficiently if u is in S, use an additional `boolean`

Search in Weighted Graphs

- Many applications require weights attached to the edges e.g., the transition probabilities
- Goal: find the shortest path
- We will look at single-source shortest path with nonnegative weights
  - The Bellman-Ford algorithm works with negative weights, too
  - For graphs with negative cycles, the shortest path is not well defined
- First: Dijkstra's algorithm for SSSP with nonnegative weights
- Generalization: A^* search with a heuristic function

Dijkstra's SSSP

- Algorithm relies on the triangle equation:
  $dist(u, w) + weight(w, v) \geq dist(u, v)$ for all $u, v, w \in \mathcal{V}$
- Initially:
  - set the distances for all nodes to $+\infty$, except for the source node s to zero
  - mark all nodes as not optimized
- While there are nodes not yet optimized:
  - take the unoptimized node u with the smallest $dist(u)$
  - check for all neighbours v if the triangle equation is violated, that is:
    $dist(u) + weight(u, v) < dist(v)$
  - if so, correct $dist(v)$ and store u as predecessor of v

Dijkstra's SSSP II

Dijkstra-SSPP(s, G)

1: **for all** $v \in \mathcal{V}$ **do**
2:     $\mathit{dist}(v) = +\infty$; $\mathit{predecessor}(v) =$undef; Q.add(v)
3: $\mathit{dist}(s) = 0$
4: **while** $Q \neq \emptyset$ **do**
5:     $u = Q$.extract_min()
6:     **for all** $(u, v) \in \mathcal{E}$ **do**
7:         $\mathit{alt} = \mathit{dist}(u) + \mathit{weight}(u, v)$
8:         **if** $\mathit{alt} < \mathit{dist}(v)$ **then**
9:             $\mathit{dist}(v) = \mathit{alt}$; $\mathit{predecessor}(v) = u$

- Finally, the predecessor chain can be followed backwards from any node for the shortest path to s
- The algorithm can be stopped in line 5 if u is the desired target node

Data Structure for Q

- Q must support the operations
  - add element
  - extract_min : get and remove the element with the lowest key
  - lower_key : lower the key of an arbitrary element
- `java.util.PriorityQueue` supports the first two efficiently

Data Structure for Q

- Q must support the operations
  - add element
  - extract_min : get and remove the element with the lowest key
  - lower_key : lower the key of an arbitrary element
- `java.util.PriorityQueue` supports the first two efficiently
- *BUT:* lower_key can only be implemented using: `remove(v)+add(v)`, which means $O(n) + O(\lg(n))$
- To avoid the search in `remove(v)`, relate the *elements* efficiently to the *buckets* of the priority queue
- To do so, a homemade priority queue is requied

A* Search

- If the search space is very big (as in most AI complete problems), Dijkstra's algorithm may be too expensive
- Use additional information to guide the search, if available
- This is mostly called rest-cost estimate or heuristic function of the search
- Will only affect the average time for finding the goal
- Incrementally explore all paths until the optimal path is found:
  - The solution is sound and complete
  - Because of the additional bookkeeping, it can get worse than the plain algorithm, but will behave better in practice

A* Search: Simple Example

- Get from Place s to t using the map of a city
  - Vertices: crossings
  - Edges: connecting roads (eventually one-way)
  - Weights: Length of the road between two crossings
  - If at crossing x, we know the $dist(x)$ already traveled
  - In addition, we have an estimate for the rest: the air-line distance between x and the target t
- Instead of using $dist(x)$ (Dijkstra), use $dist(x) + airline(x, t)$ as weight for the priority queue
- If the remaining cost are never overestimated, the heuristic is admissible and the optimum will be found
- Dijkstra is a special A*, with the rest cost estimate zero

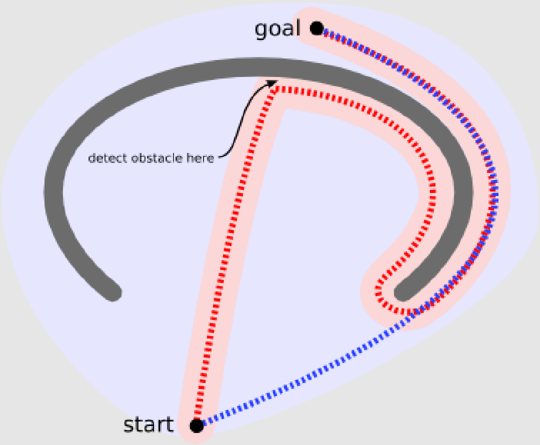
Automatic Pathfinding

Task: Move an object through an environment with obstacles from a start to a goal location

- find the shortest path
- find the fastest path

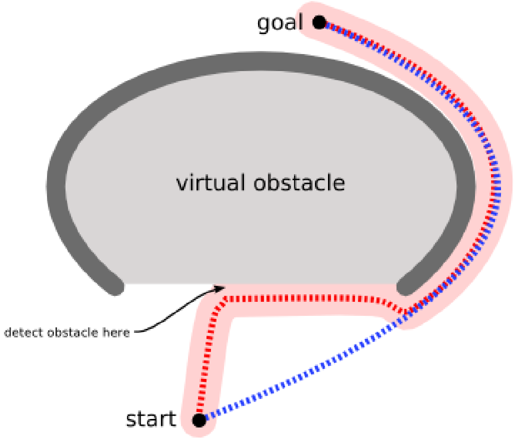
Problem Desription

Concave obstacles pose a severe problem



Obstacle Avoidance

- Compute convex hulls beforehand and avoid entering them
- Design algorithm to handle this problem

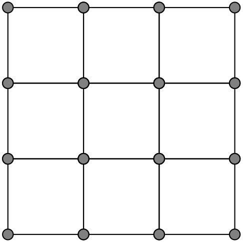


Problem Formulation as Graph

- Map the plane onto a grid, and connect nodes
- Distance between nodes?

Problem Formulation as Graph

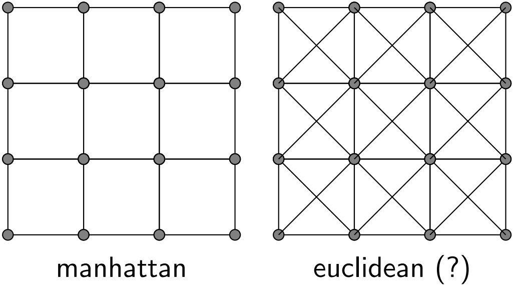
- Map the plane onto a grid, and connect nodes
- Distance between nodes?



manhattan

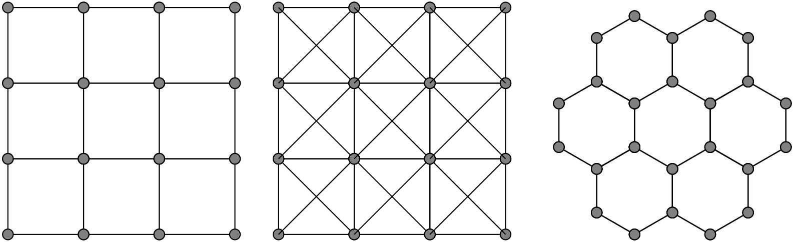
Problem Formulation as Graph

- Map the plane onto a grid, and connect nodes
- Distance between nodes?



Problem Formulation as Graph

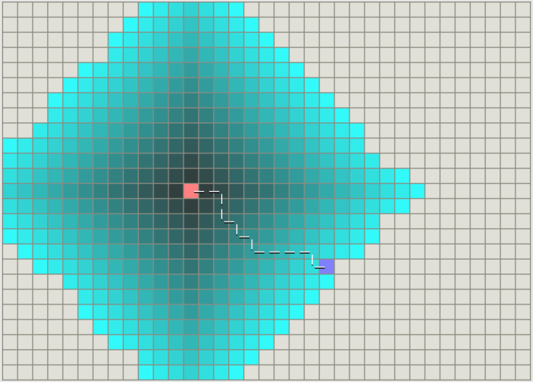
- Map the plane onto a grid, and connect nodes
- Distance between nodes?



manhattan   euclidean (?)   euclidean

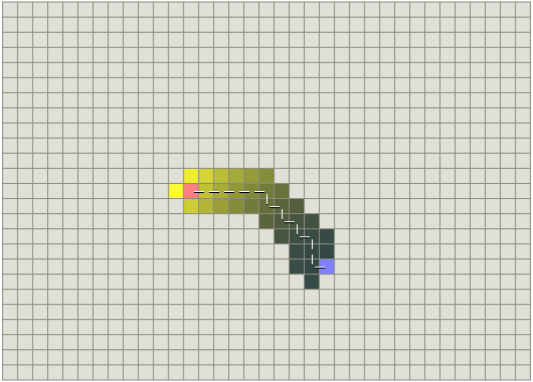
Dijkstra on a Rectangular Grid

Dijkstra's algorithm visits the non-visited node nearest to the starting point first



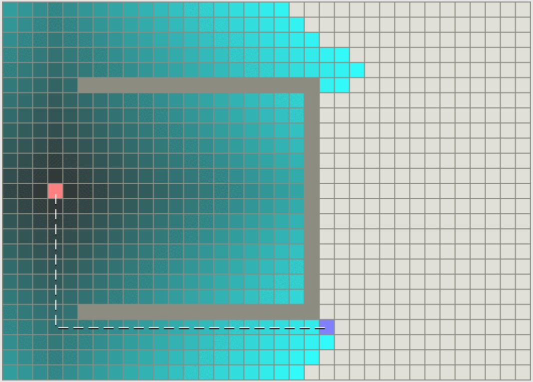
Greedy Best-First Search

- Needs additional information about the graph, such as relative position of current and goal node, a rest-cost estimate, like A*
- Always takes the node with the best heuristic value first
- Can significantly improve over Dijkstra



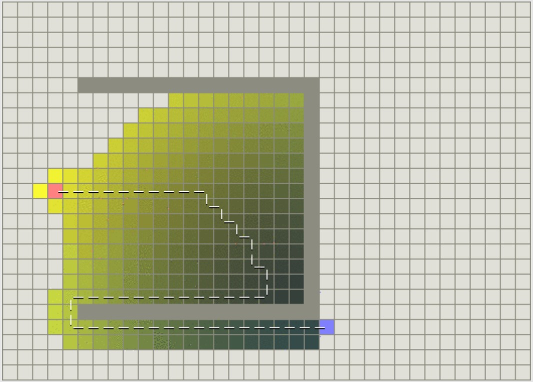
Dijkstra Behaviour With Obstacles

Guaranteed to find shortest path, even with obstacles



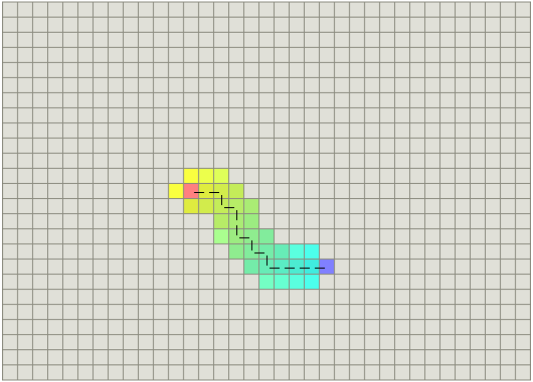
Greedy Best-First With Obstacles

Less nodes visited, but suboptimal path



A* as combination of Dijkstra and Best-First

- A* combines information from Dijkstra (distance from start) with estimated information about the remaining cost (like best-first)
- Without obstacles, A* behaves like greedy best-first



Improved Behaviour of A*

- The combination of information avoids search in implausible directions
- Better performance coupled with good results, depending on the accuracy of the rest-cost estimation