Faculty of Mathematics and Physics
Charles University
October 22, 2024

Artificial Intelligence for Computer Games

Local Navigation (continued)
Grid-Based Pathfinding

Adam Dingle

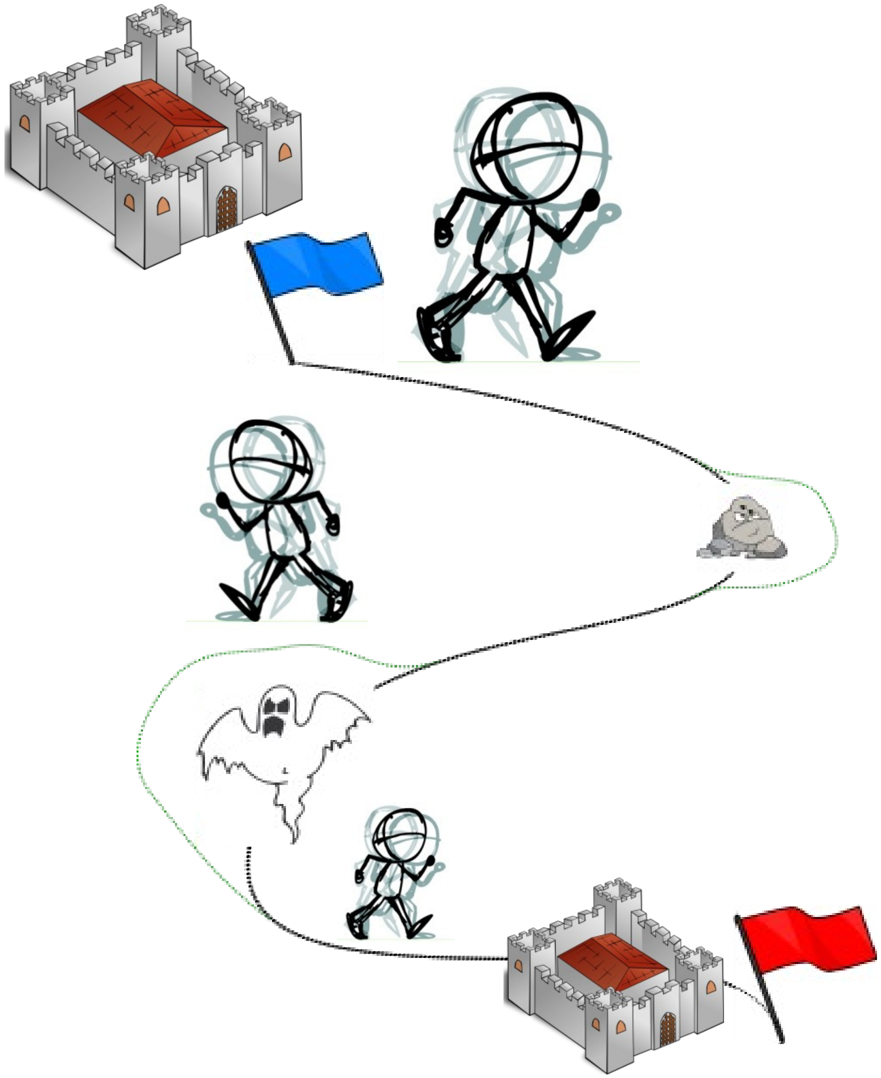
Steering Behaviors
As part of navigation

1. Action-Selection
Strategy => Goals

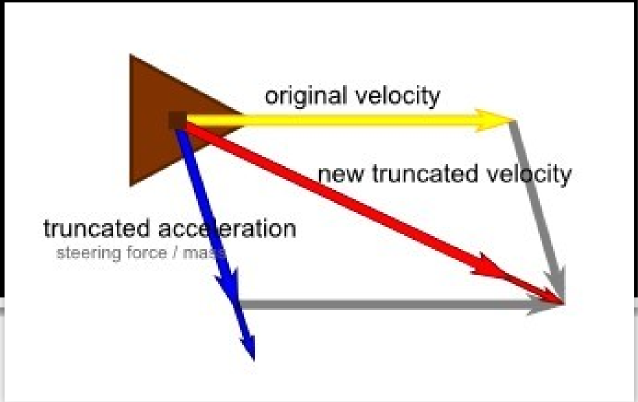
2. Path-planning
List of path-points

3. Path-determination or Path-following
What path to take exactly

4. Animating
Animation sequencing



Steering Behaviors
Vehicle Model




```
1. accel = steering.calculate(args)
2. accel = truncate(accel, max_accel)
3. velocity = velocity + accel * timeDelta
4. velocity = truncate(velocity, max_speed)
5. position += velocity * timeDelta
6. look-direction = velocity.normalized
```

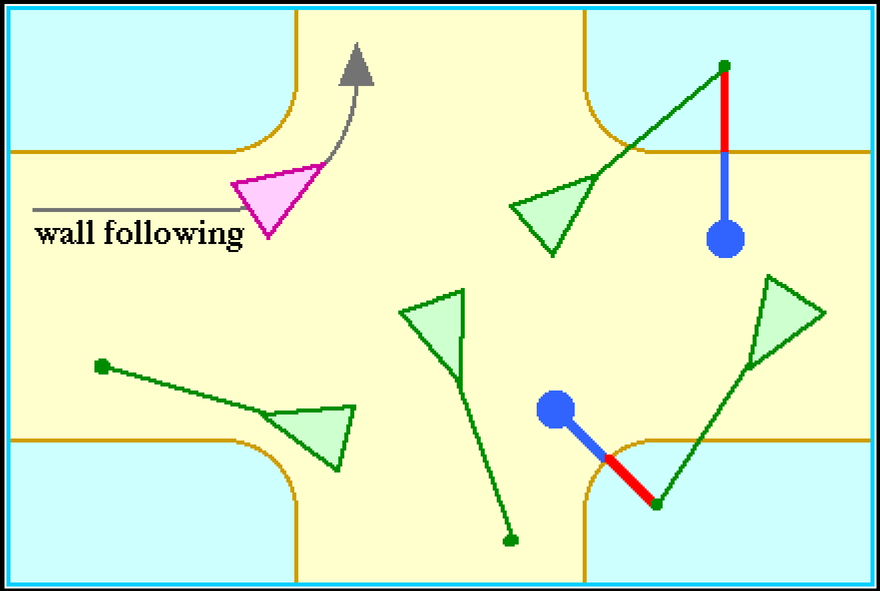
Steering Behaviors
List of Reynolds steerings

- Simple behaviors for individuals and pairs:
  - Seek and Flee (static target)
  - Pursue and Evade (moving target)
  - Wander
  - Arrive
  - Path Following
  - Wall Following
  - Containment
  - Obstacle Avoidance

- Combined behaviors and groups:
  - Flocking (combining: separation, alignment, cohesion)
  - Leader Following
  - Crowd Path Following
  - Unaligned Collision Avoidance

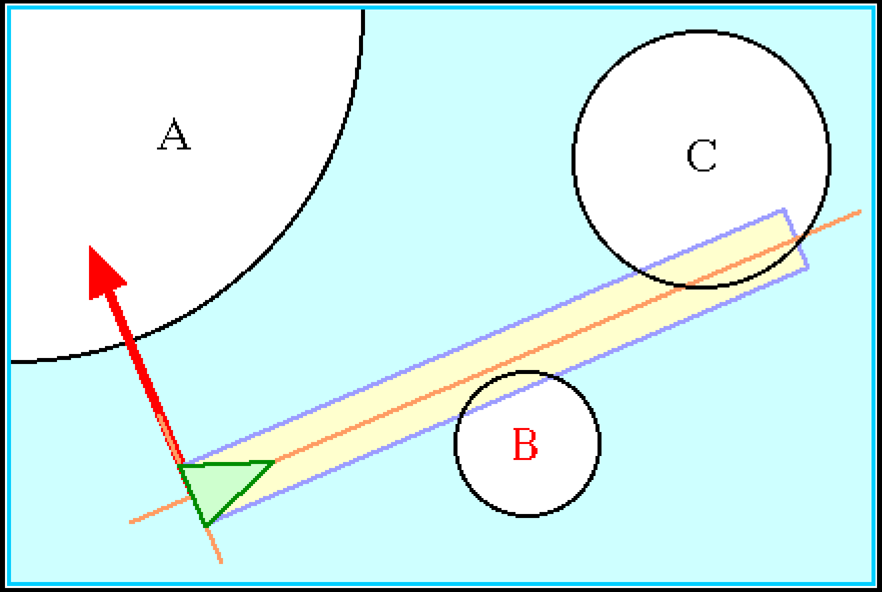
Craig Reynolds, "Steering Behaviors For Autonomous Characters" (1999)

Wall Following
Containment



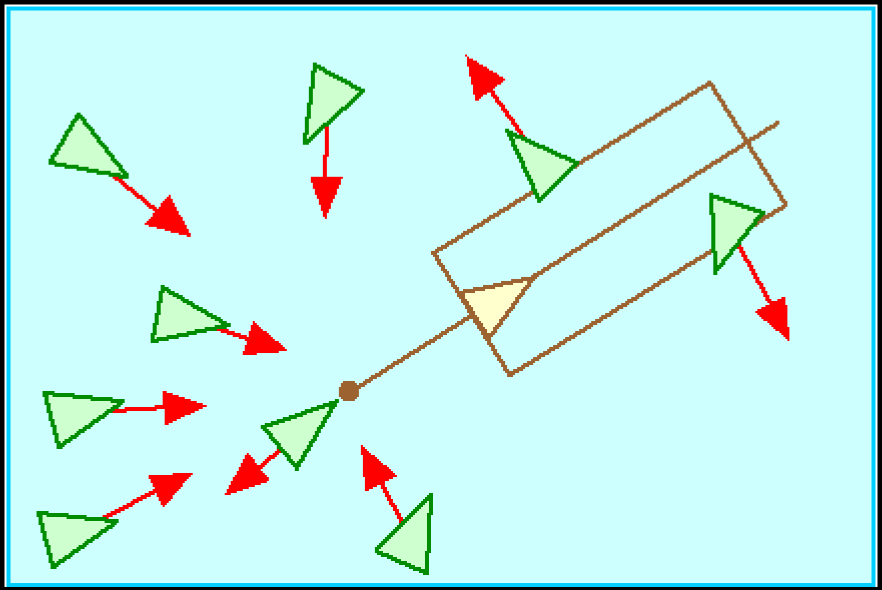
Obstacle Avoidance

- Goal: keep empty cylinder ahead
- Detect nearest obstacle that will collide
- Accelerate laterally away



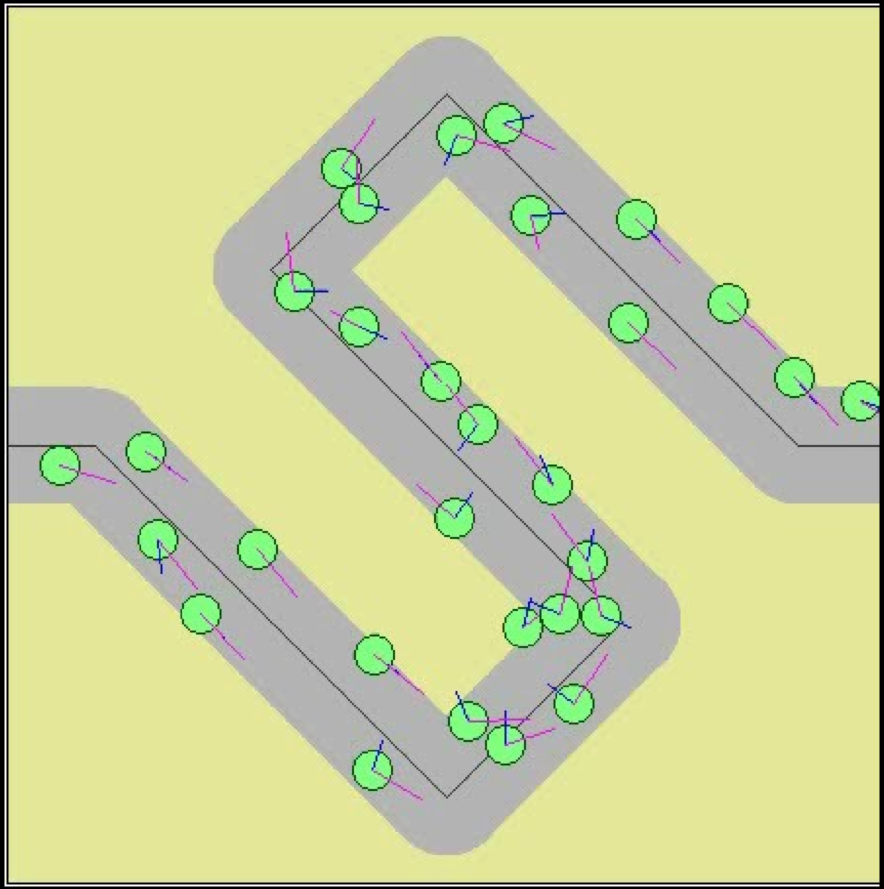
Leader Following

- Agents are steered to follow a leader
- Steering force consists of:
  - **Arrival** – the target is slightly behind leader
  - **Separation** – to prevent collisions with other followers
  - A follower in a rectangular region in front of the leader will steer away from the leader's path



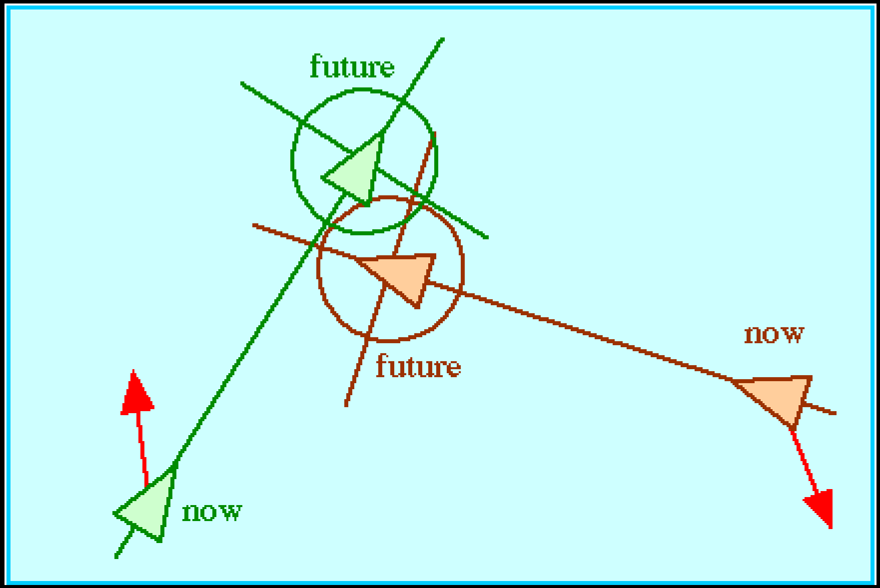
Crowd Path Following

- Path Following + Separation



Unaligned Collision Avoidance

- Predict next potential collision
- Steer laterally to turn away
- Can also accelerate or decelerate

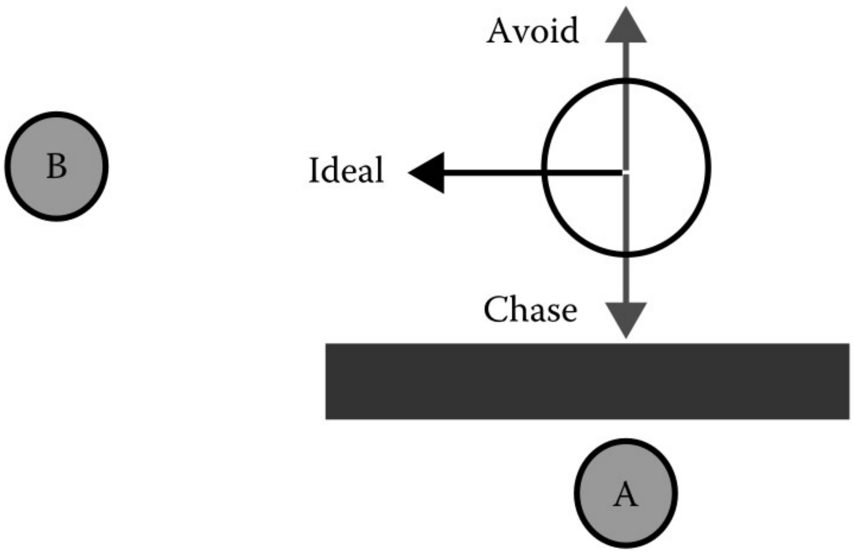


Steering Behaviors
Combining Behaviors

- Can add behaviors
  - possibly with weighting factor
- Can prioritize behaviors, e.g.
  1. avoid obstacle if nearby
  2. evade enemy if nearby
  3. seek to goal

Steering Behaviors
Context Steering

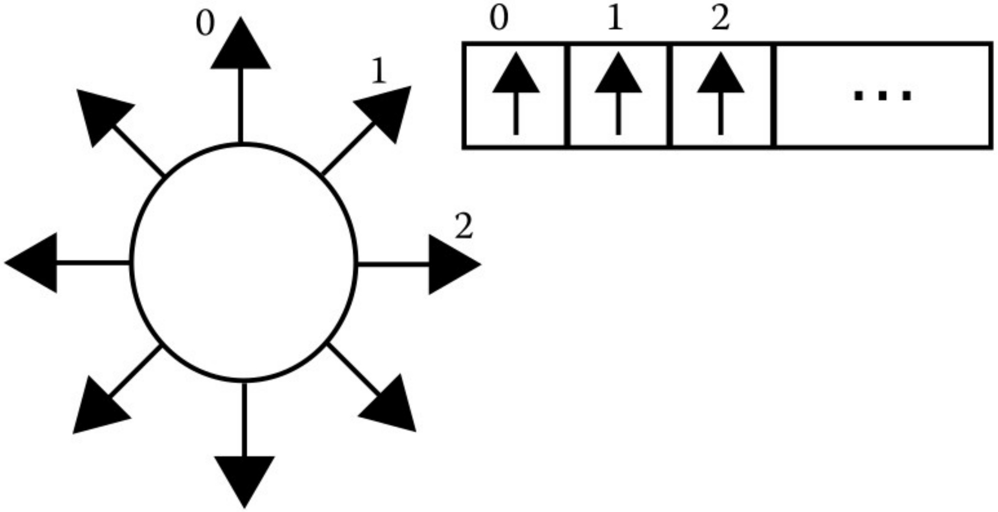
- Problem: sum of steering behaviors is not always ideal



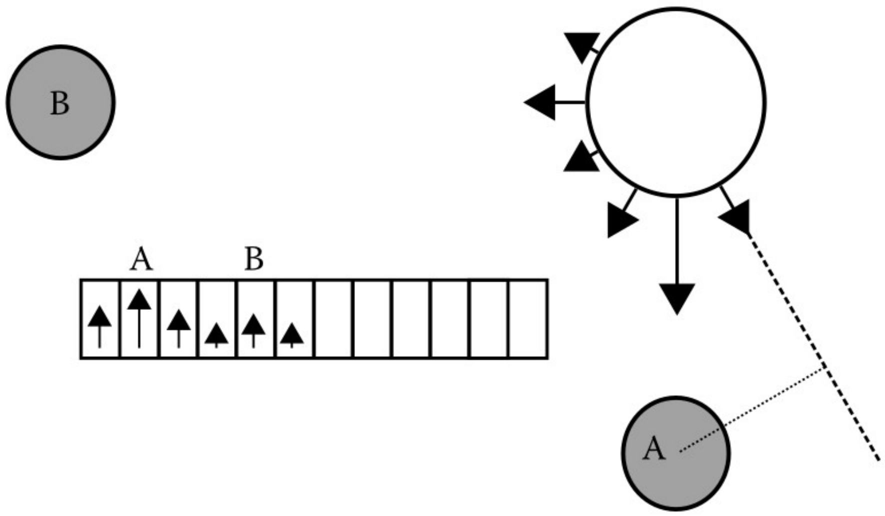
Andrew Fray, "Context steering: Behavior-driven steering at the macro scale" (2015)

Steering Behaviors
Context Steering

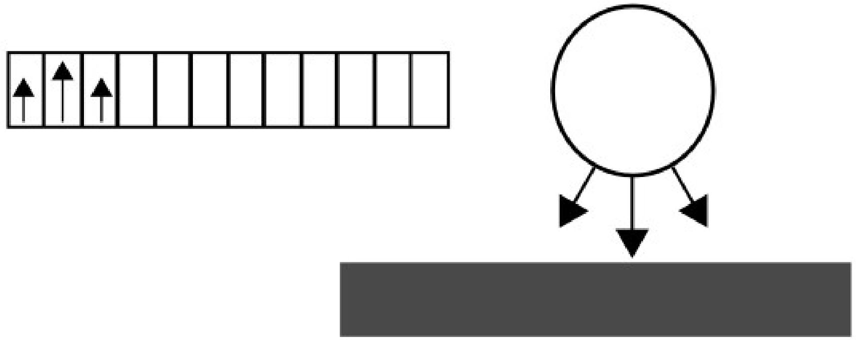
- Solution: generate **context maps**
- Map directions to slots



Steering Behaviors
Context Steering



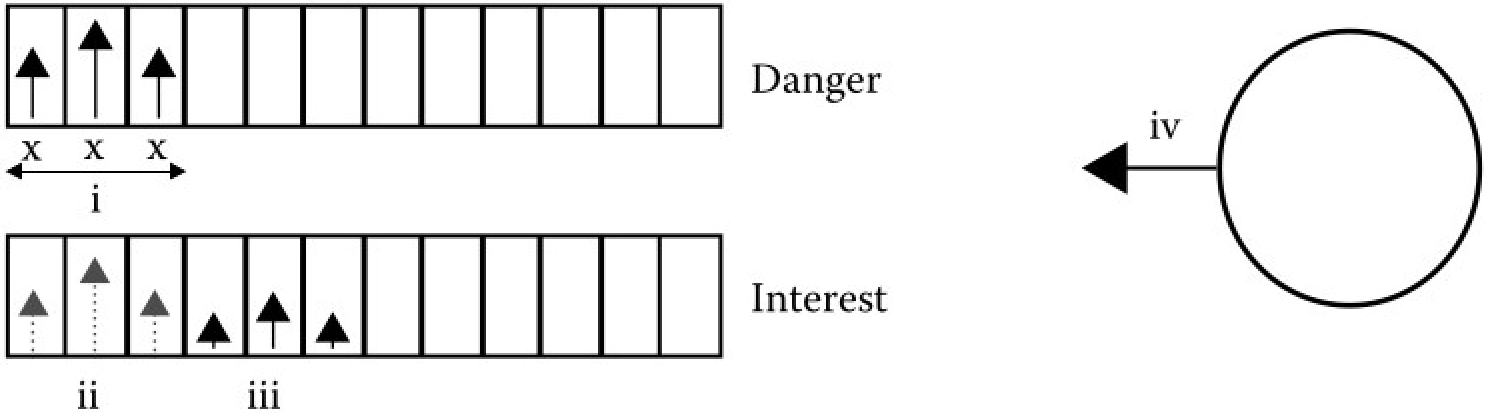
Store chase behavior in **interest map**



Store avoid behavior in **danger map**

Steering Behaviors
Context Steering

- Find slots with lowest danger
- From those, choose slot with highest interest



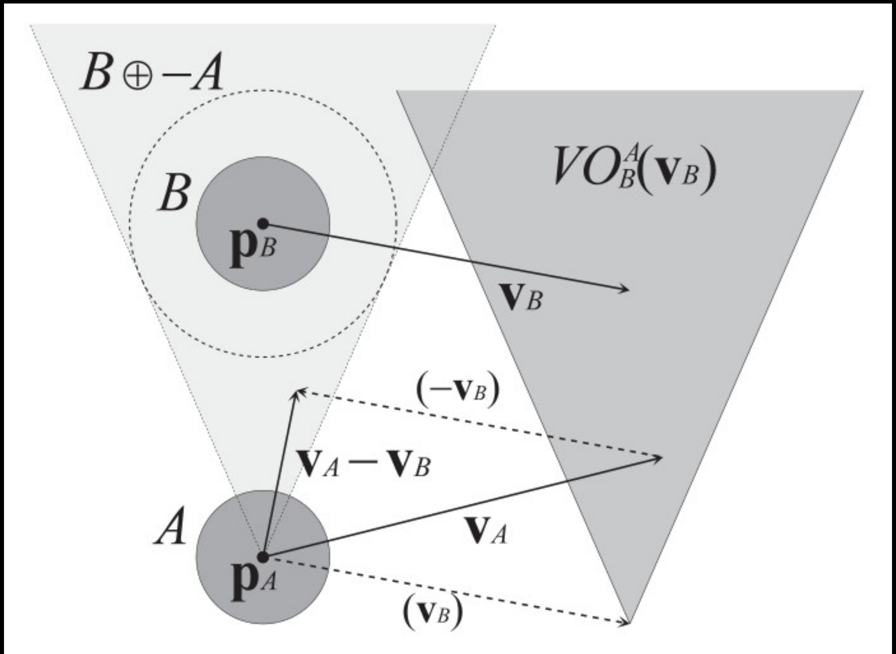
Velocity Obstacles

- Goal: avoid collisions more reliably than we can achieve with steering behaviors
- Invented in robotics (and elsewhere)

P. Fiorini and Z. Shiller, "Motion planning in dynamic environments using velocity obstacles" (1998)

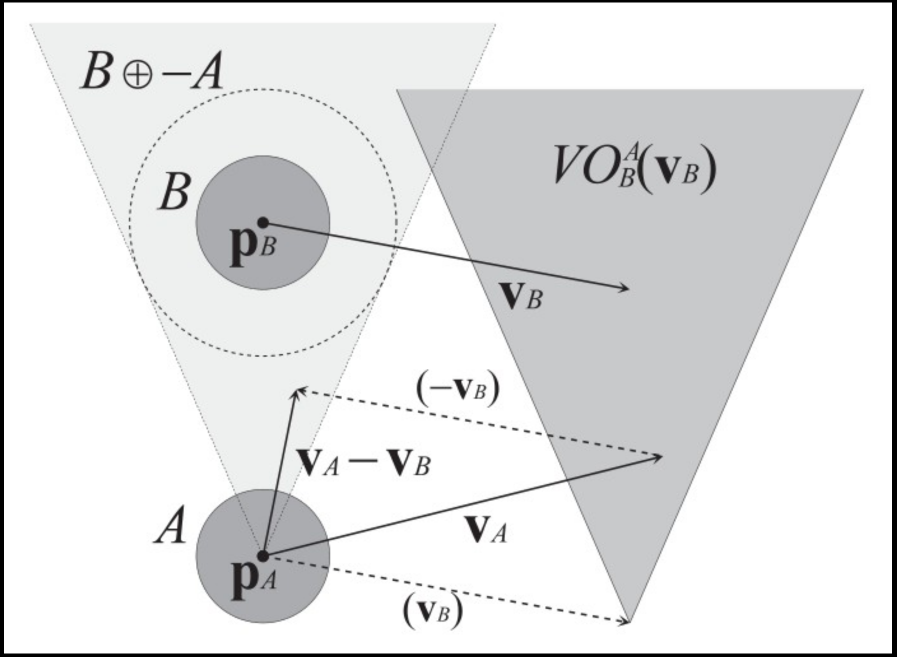
Velocity Obstacles

- Agents *A*, *B* are at positions $\mathbf{p}_A$, $\mathbf{p}_B$
- *B* is travelling at a fixed velocity $\mathbf{v}_B$
- *Which velocities* $\mathbf{v}_A$ *will allow* A *to avoid colliding with* B*?*



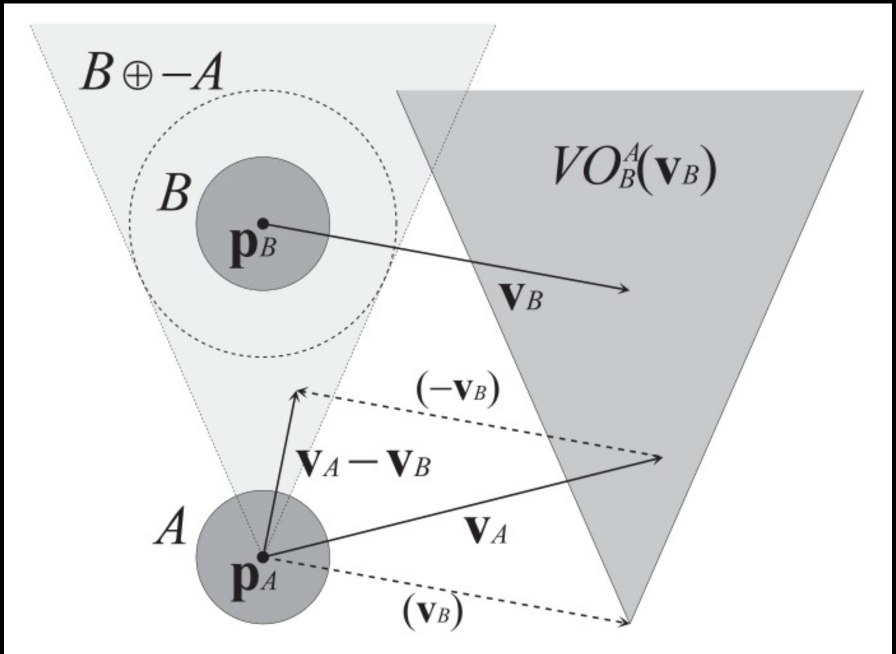
Velocity Obstacles

- $B \oplus -A$ is the set of positions where A would collide with B
- $S \oplus T = \{\, \mathbf{s} + \mathbf{t} \mid \mathbf{s} \in S, \mathbf{t} \in T \,\}$ (Minkowski sum)
- If A and B are circles, $B \oplus -A$ is a circle whose radius is the sum of the radii of A and B



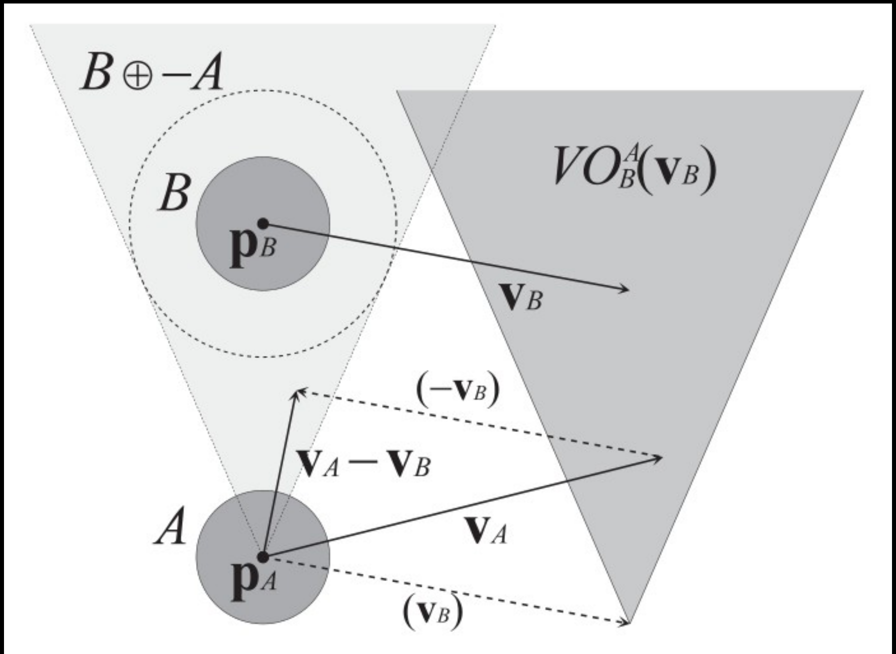
Velocity Obstacles

- The left cone shows velocities $\mathbf{v}_A$ for which *A* would not collide with *B if* B *were not moving*
- But *B* is moving at $\mathbf{v}_B$, so we must consider the *relative velocity* $\mathbf{v}_A - \mathbf{v}_B$



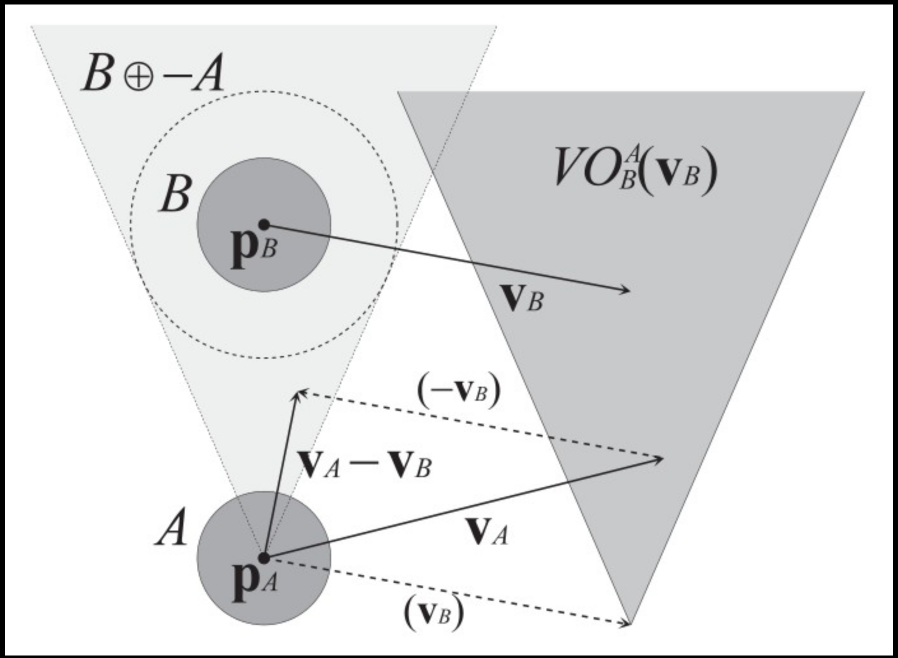
Velocity Obstacles

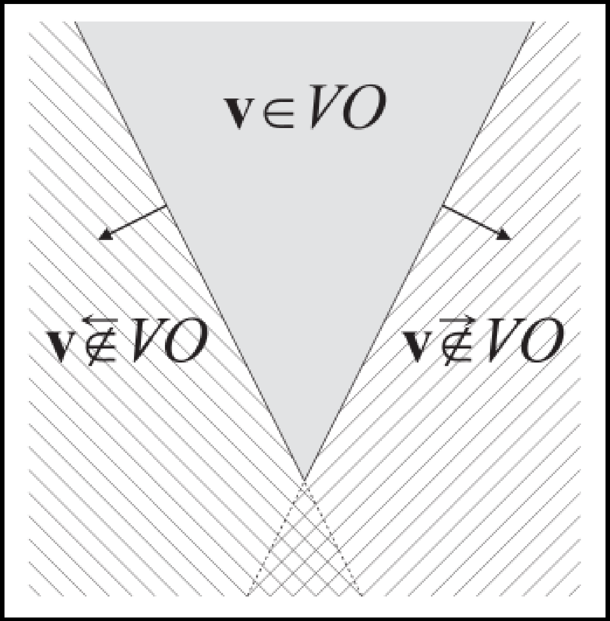
- $\lambda(\mathbf{p}, \mathbf{v}) = \{ \mathbf{p} + t\mathbf{v} \mid t \geq 0 \}$
  - ray starting at p, heading in direction v
- $VO^A{}_B(\mathbf{v}_B) = \{ \mathbf{v}_A \mid \lambda(\mathbf{p}_A, \mathbf{v}_A - \mathbf{v}_B) \cap (B \oplus -A) \neq \emptyset \}$
  - velocity obstacle of *B* to *A*
  - set of velocities $\mathbf{v}_A$ for which *A* will collide with *B*!



Velocity Obstacles

- VO^A_B is a cone with apex at v_B
- If $v_A \in VO^A_B$, A will collide with B
- If v_A is the apex velocity v_B, they will not collide
- If v_A is in the left or right half-plane outside VO^A_B, A will pass B on the left or right

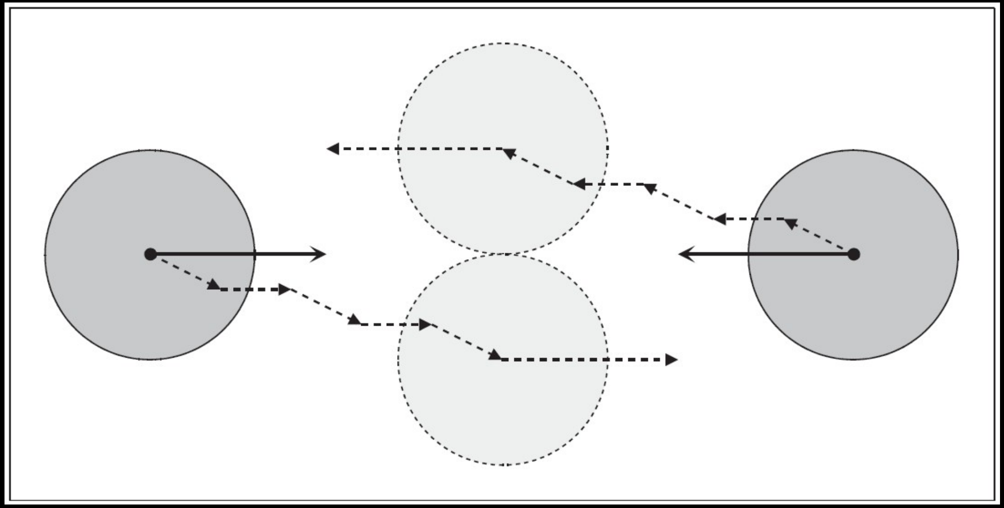




Reciprocal Velocity Obstacles

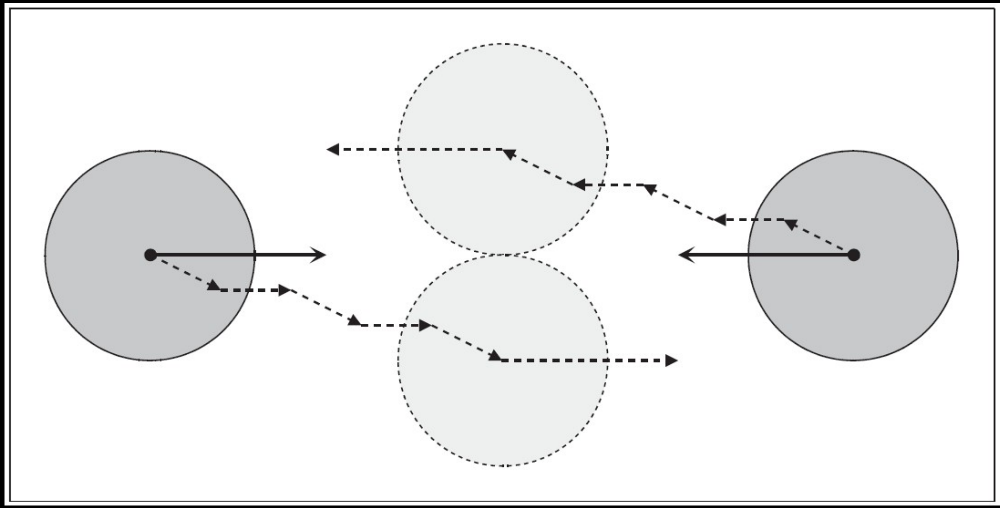
Extension of Velocity Obstacles

- Suppose *A* and *B* have velocities $\mathbf{v}_A$ and $\mathbf{v}_B$, and are on a collision course
- They choose new velocities $\mathbf{v}'_A \notin VO^A_B(v_B)$, $\mathbf{v}'_B \notin \mathrm{VO}^B_A(\mathbf{v}_A)$
  - They must choose to pass on the same side (or may still collide)



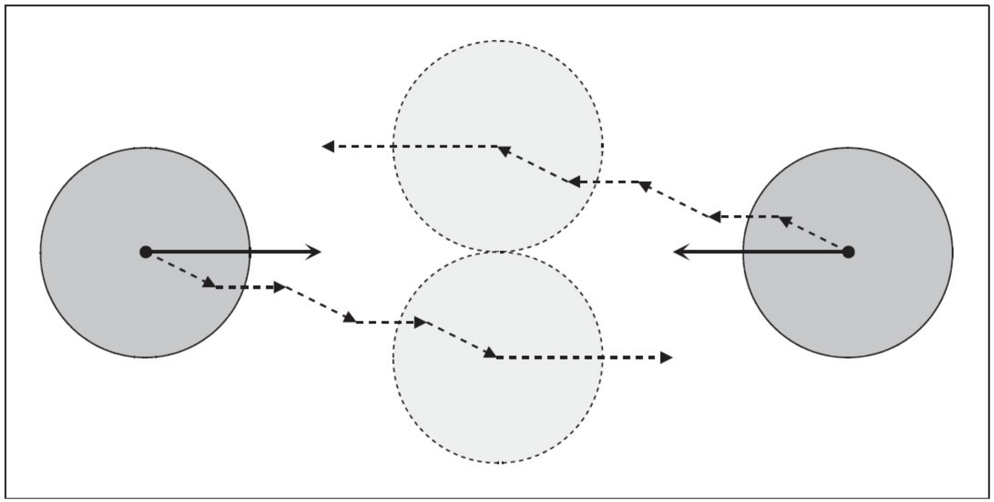
Reciprocal Velocity Obstacles

- The old velocities $\mathbf{v}_A$ and $\mathbf{v}_B$ will be outside the new velocity obstacles $VO^A_B(v'_B)$ and $VO^B_A(\mathbf{v'}_A)$
- So the agents may immediately switch back to $\mathbf{v}_A$ and $\mathbf{v}_B$ if they prefer those velocities
- The velocities will oscillate!



Reciprocal Velocity Obstacles

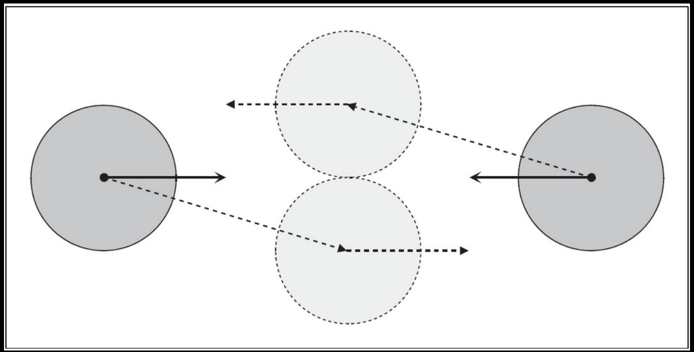
- Reciprocal velocity obstacles will let the agents pass each other naturally without oscillation



Van den Berg et al, "Reciprocal velocity obstacles for real-time multi-agent navigation" (2008)

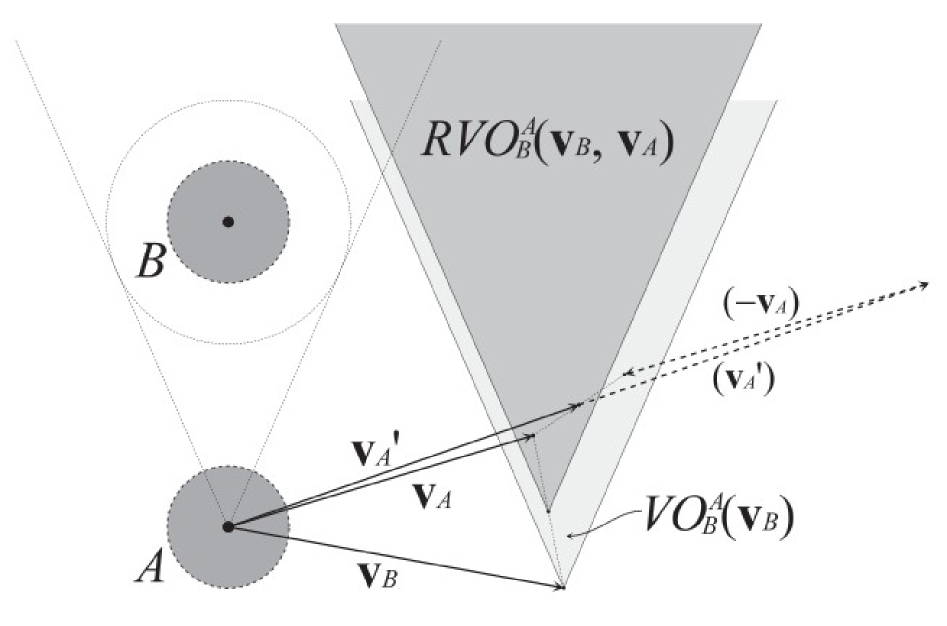
Reciprocal Velocity Obstacles

- Basic idea: each agent will choose a velocity that goes only *halfway* toward resolving the collision
- So $\mathbf{v}'_A = (\mathbf{v}_A + \mathbf{v}) / 2$ for some $\mathbf{v} \notin VO^A_B(\mathbf{v}_B)$
  - and so $2\mathbf{v}'_A - \mathbf{v}_A \notin VO^A_B(\mathbf{v}_B)$
- Definition (Reciprocal Velocity Obstacle):
  - $RVO^A_B(\mathbf{v}_B, \mathbf{v}_A) = \{\mathbf{v}'_A \mid 2\mathbf{v}'_A - \mathbf{v}_A \in VO^A_B(\mathbf{v}_B) \}$



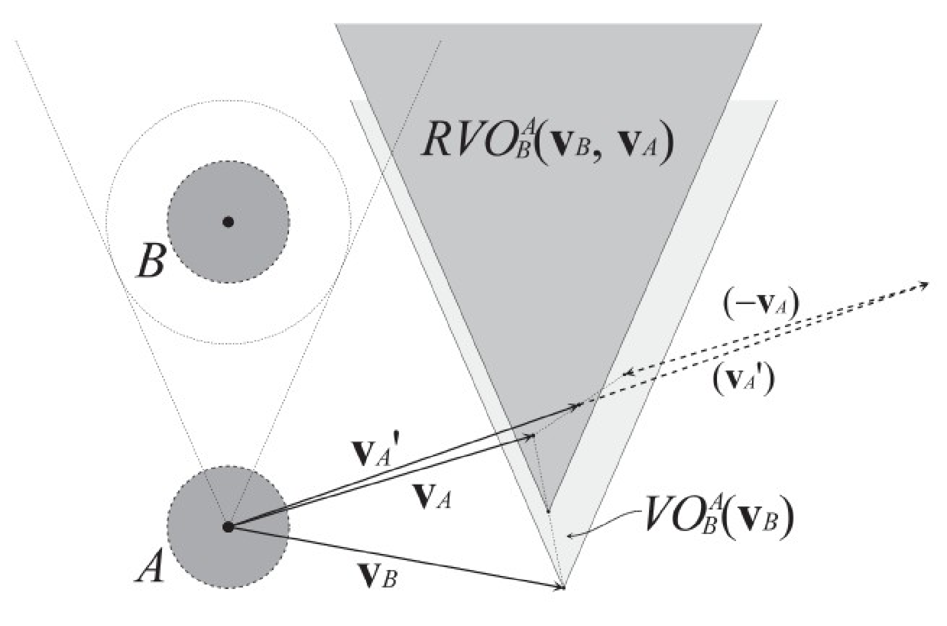
Reciprocal Velocity Obstacles

- $RVO^A_B(\mathbf{v}_B, \mathbf{v}_A) = \{\mathbf{v}'_A \mid 2\mathbf{v}'_A - \mathbf{v}_A \in VO^A_B(\mathbf{v}_B)\}$
- A cone with apex $(v_A + v_B) / 2$
- If both A and B choose the apex velocity, they will move at the same speed and won't collide



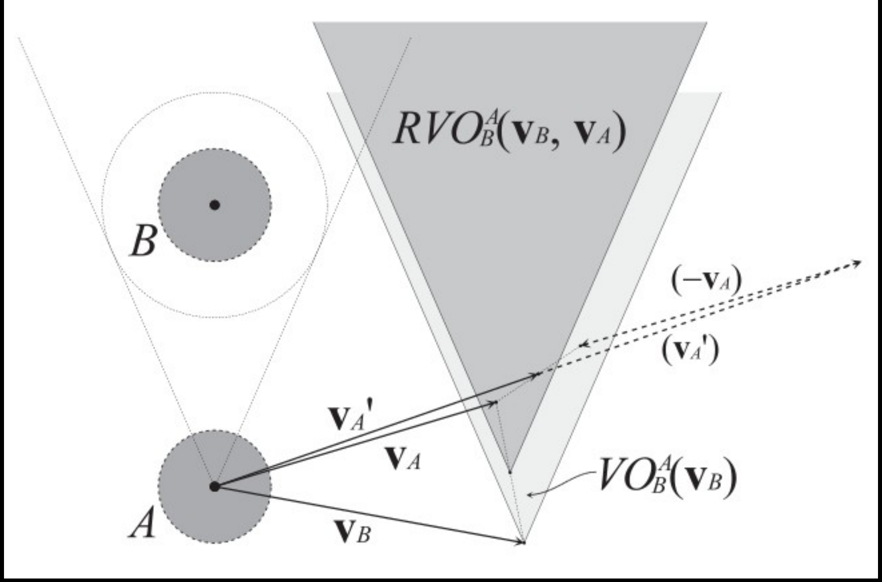
Reciprocal Velocity Obstacles

- Theorem 1: If *A* and *B* choose new velocities $\mathbf{v}'_A$ and $\mathbf{v}'_B$ outside each other's RVO, they will not collide, as long as they choose to pass on the same side
  - $\mathbf{v}'_A \notin RVO^A{}_B(\mathbf{v}_B, \mathbf{v}_A) \wedge \mathbf{v}'_B \notin RVO^B{}_A(\mathbf{v}_A, \mathbf{v}_B) \Rightarrow \mathbf{v}'_A \notin VO^A{}_B(\mathbf{v}'_B) \wedge \mathbf{v}'_B \notin VO^B{}_A(\mathbf{v}'_A)$
  - Proof: easy algebra



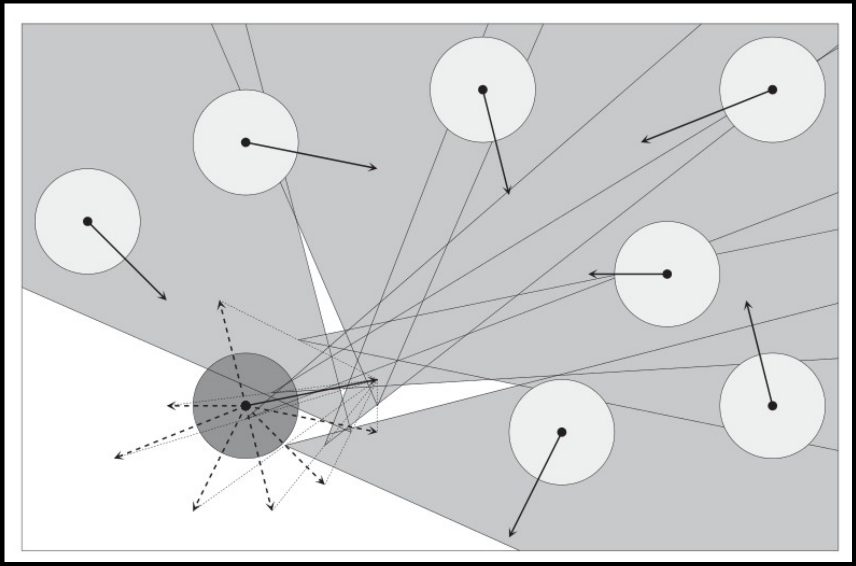
Reciprocal Velocity Obstacles

- Theorem 2: Suppose that *A* chooses a new velocity $\mathbf{v}'_A$ that is outside *B*'s RVO and *as close as possible* to $\mathbf{v}_A$, and *B* similarly chooses $\mathbf{v}'_B$. Then
  1. *A* and *B* will choose the same side to pass each other.
  2. $\mathbf{v}_A \in RVO^A_B(\mathbf{v}'_B, \mathbf{v}'_A)$ (and similarly for $\mathbf{v}_B$).
  3. $RVO^A_B(\mathbf{v}'_B, \mathbf{v}'_A) = RVO^A_B(\mathbf{v}_B, \mathbf{v}_A)$: the RVO does not change (and similarly for $RVO^B_A(\mathbf{v}'_A, \mathbf{v}'_B)$).
- By (2), *A* cannot switch back to $\mathbf{v}_A$ (and similarly for *B*). No oscillations will occur.



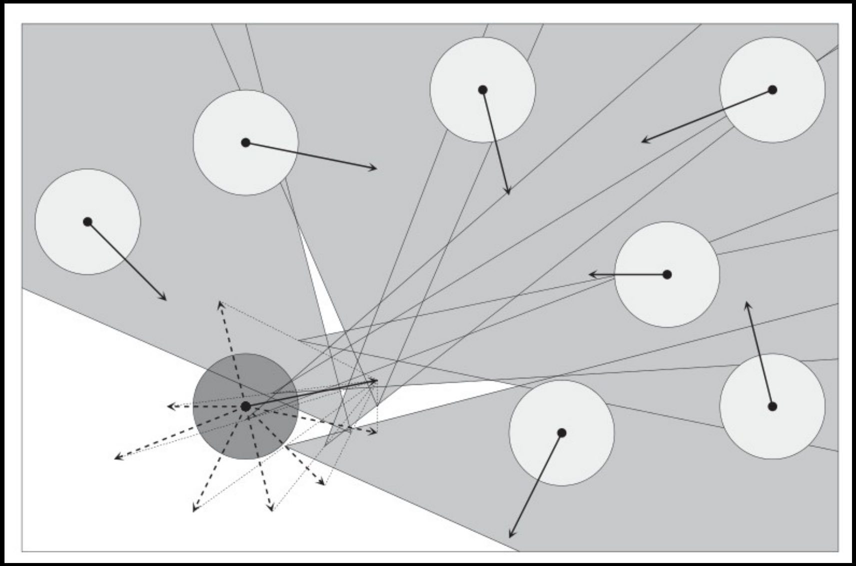
Reciprocal Velocity Obstacles

- Can also be used to avoid collisions among many agents!
- The *combined RVO* for an agent A is the union of the individual RVOs of all other agents to agent A.
- On each tick, each agent is assigned a velocity outside its combined RVO.



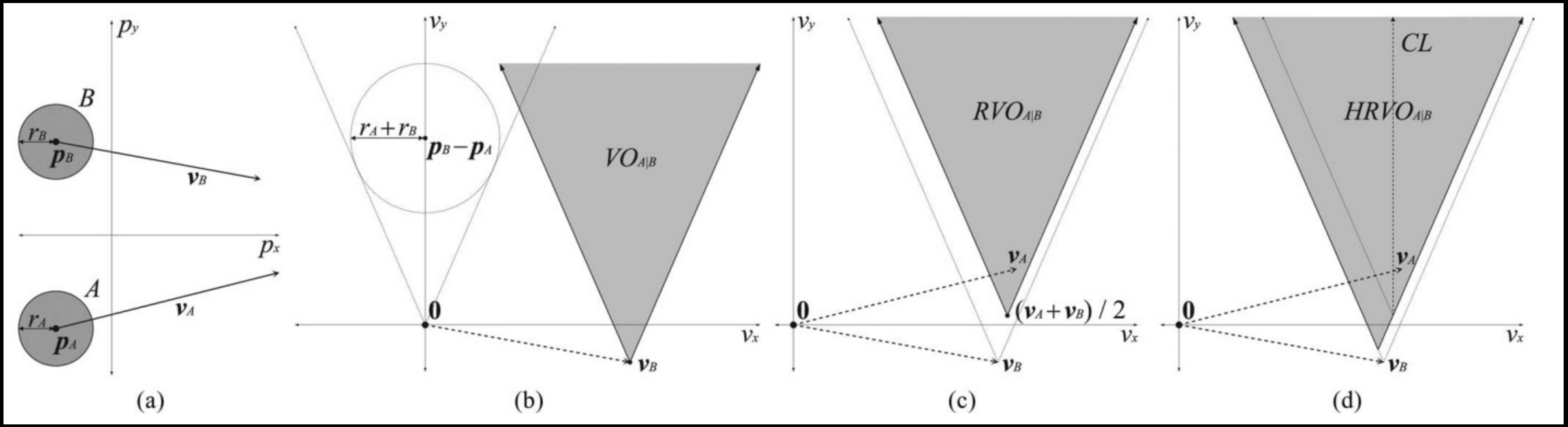
Reciprocal Velocity Obstacles

- The space may become so crowded that all admissible velocities for an agent are inside the combined RVO
- Then we choose a velocity $\mathbf{v}'$ in the combined RVO that minimizes the penalty $w \,/\, tc(\mathbf{v}') + \|\mathbf{v}^{pref} - \mathbf{v}'\|$, where
  - $\mathbf{v}^{pref}$ is the agent's preferred velocity
  - $tc(\mathbf{v}')$ is the expected time to collision with any other agent
  - w is a weighting factor



Reciprocal Velocity Obstacles

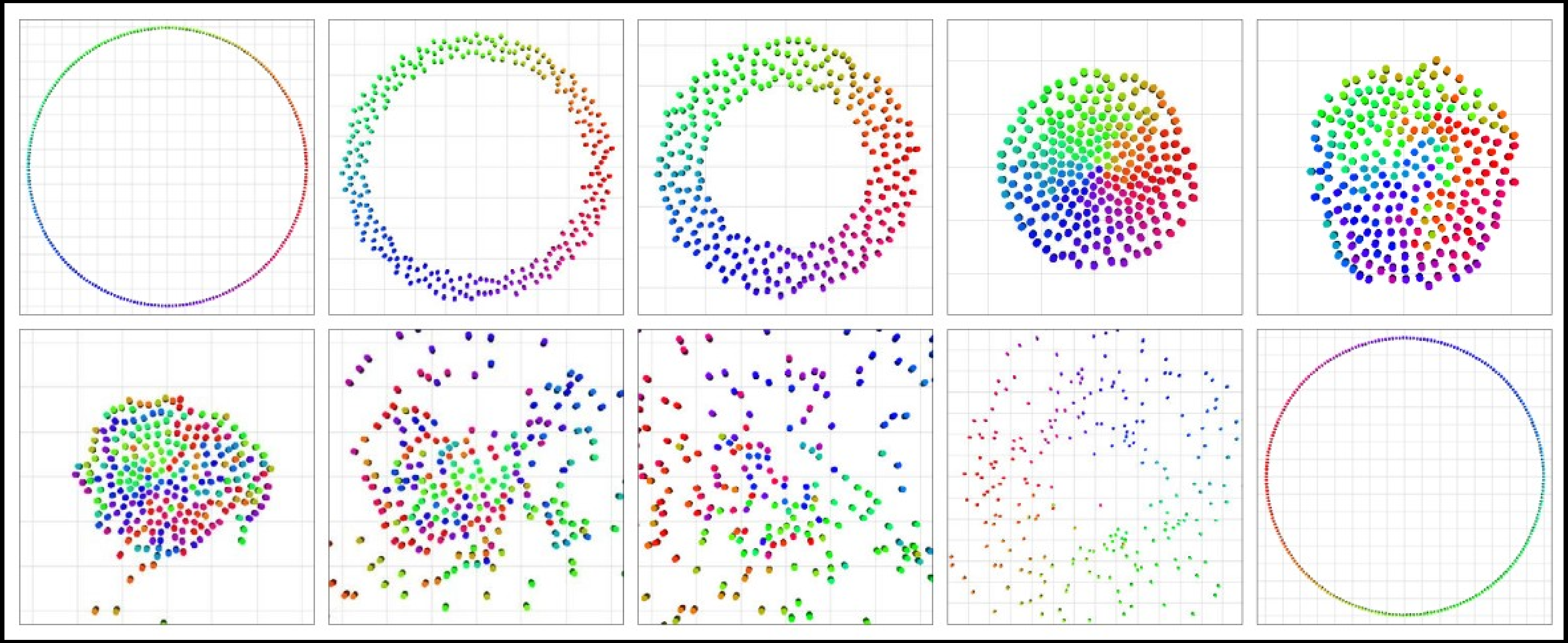
- Combining RVOs had some problems in the original RVO paper (2008)
  - Agents couldn't agree which side to pass on
- Improved using *hybrid reciprocal velocity obstacles* (2011)



Snape et al, The Hybrid Reciprocal Velocity Obstacle (2011)

Reciprocal Velocity Obstacles

- Implemented in Godot (and probably other game engines)
- Videos at
  - http://gamma.cs.unc.edu/RVO



Pathfinding
Definition of the problem

Given an environment abstraction find a good path between given start and end points for an agent.

Environment abstraction
- Tiles
- Navigation graph
- Navmesh
- Hierarchies

Start/Endpoints
- 1:1, 1:N, N:1, N:M
- Position
- Graph node

A path
- List of cells to go through
- Straight lines (list of points)
- Curves

A "good" path
- Shortest
- Cheapest
- Safest
- Believable
- ...

A **“good” path** can be quite complex…

In a dungeon complex within an asteroid, fly to enemy base

- as fast as possible
- while using as less fuel as possible
- avoid too narrow passages
- not looking mechanical
- pick as many items along the way as possible
- and avoid guard routes of the enemy.

…nd a good …nts for IVA.

…list of points)

- Tiles
- Navmesh
- Hierarchies

Start/Endpoints

- 1:1, 1:N, N:1, N:M
- Position
- Graph node
- Procedurally described

A “good” path

- Shortest
- Cheapest
- Safest
- Believable
- Most desirable
- Different then
- …

Solution of the pathfinding problem
Is…

Environment abstraction

+

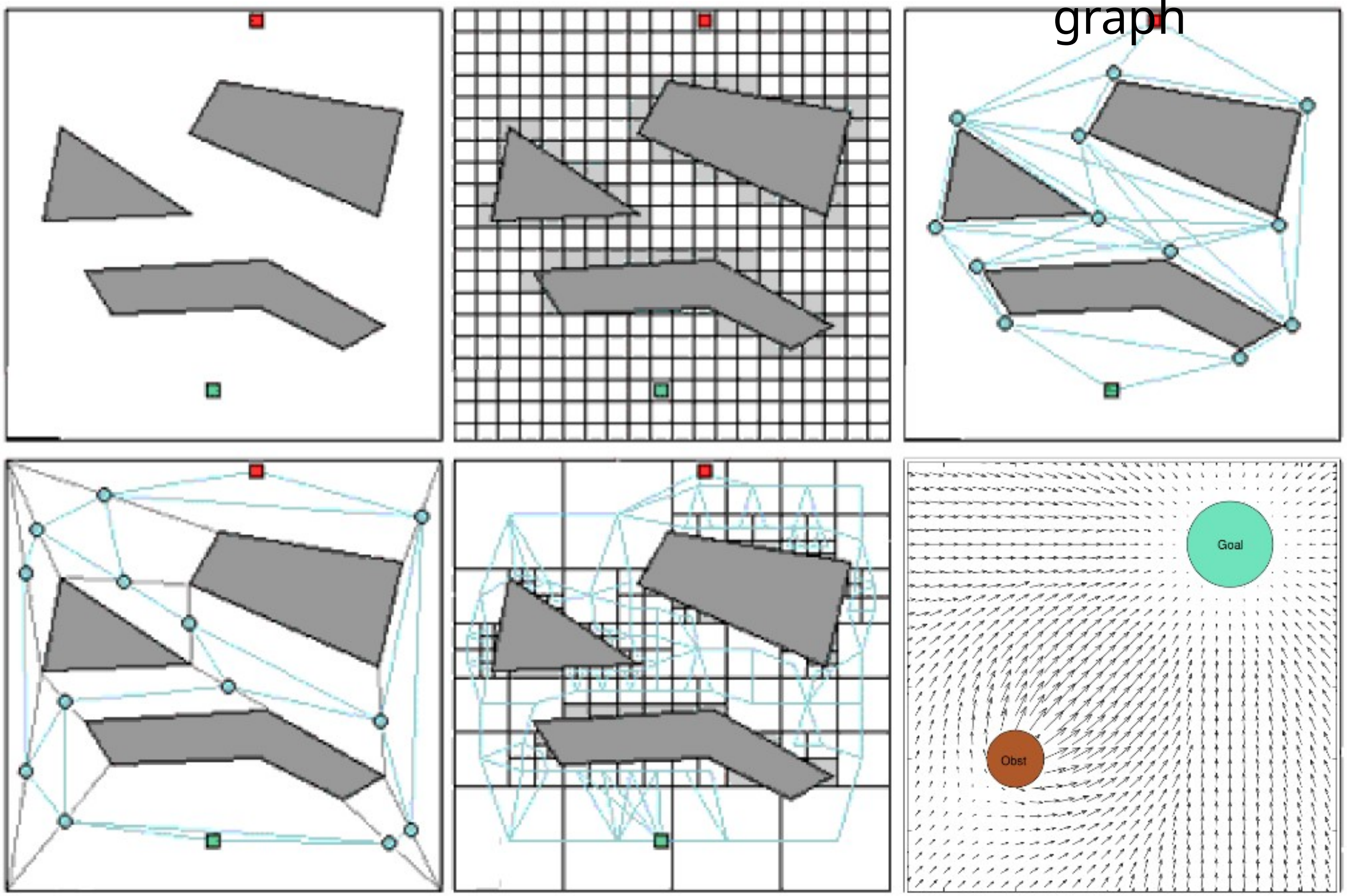
Search algorithm

that

may require **additional precomputed data** to work.

Pathfinding
Environment Abstractions

No abstraction | Rect. tiles | Navigation graph



Navmesh | Quad-tree tiles | Potential fields

Pathfinding
Solution quality metrics

Given a pathfinding solution (an environment abstraction + a pathfinding algorithm), we may consider its

Time complexity

Space complexity

Path optimality

First move delay

Precomputation requirements

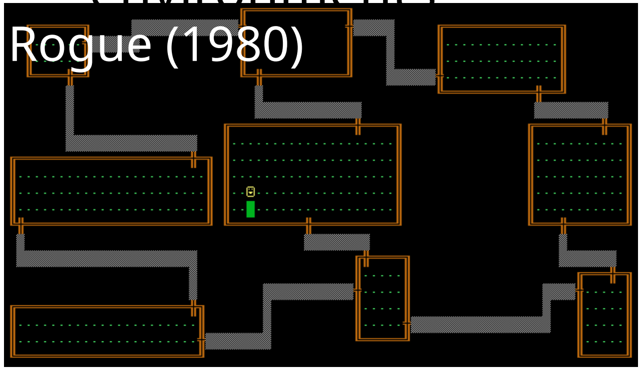
Implementation complexity

Pathfinding
Environment Abstractions

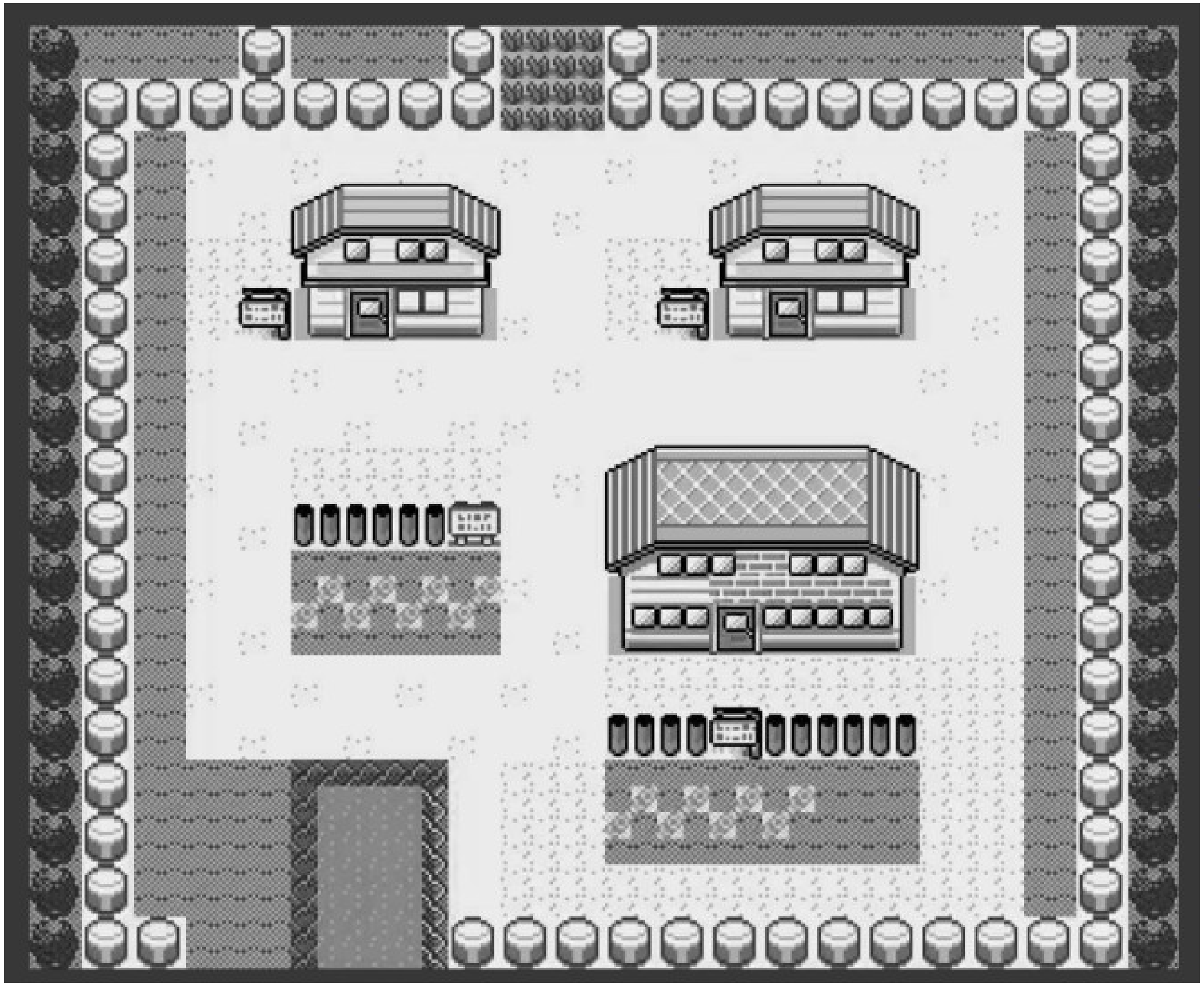
Tile-based Approaches

Environment abstraction
Regular tiles

Regular tiles
- Games started of with grid base environments


Rogue (1980)

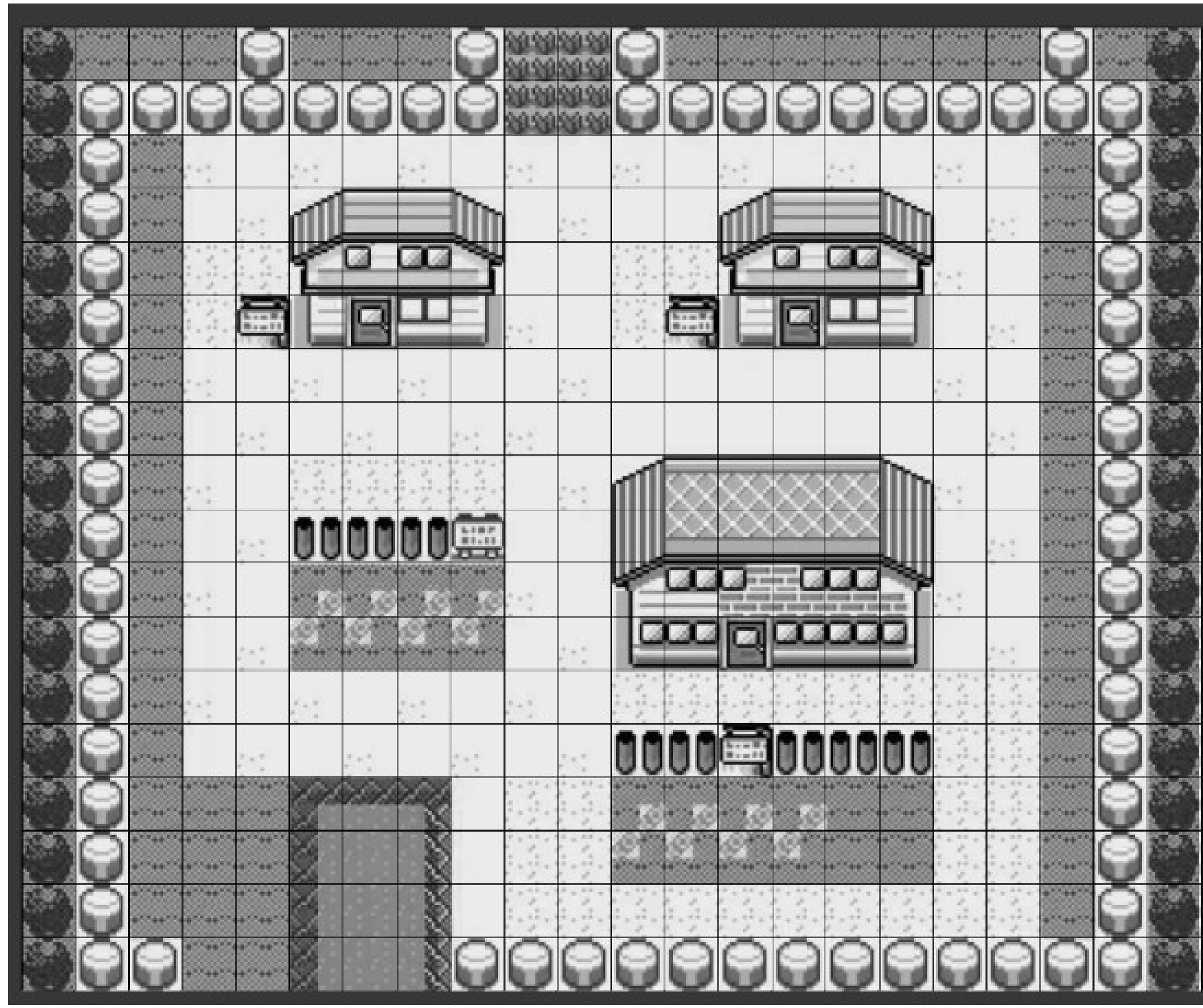

Dwarf Fortress (2006)



http://fightingkitten.webcindario.com/?p=8

Environment abstraction
Regular tiles

Regular tiles
- Some games are already grid-based
- Space is represented as a grid of regular (square or hex) tiles
- Tiles are as big as the smallest character in a game (~ smallest sprite dimension)

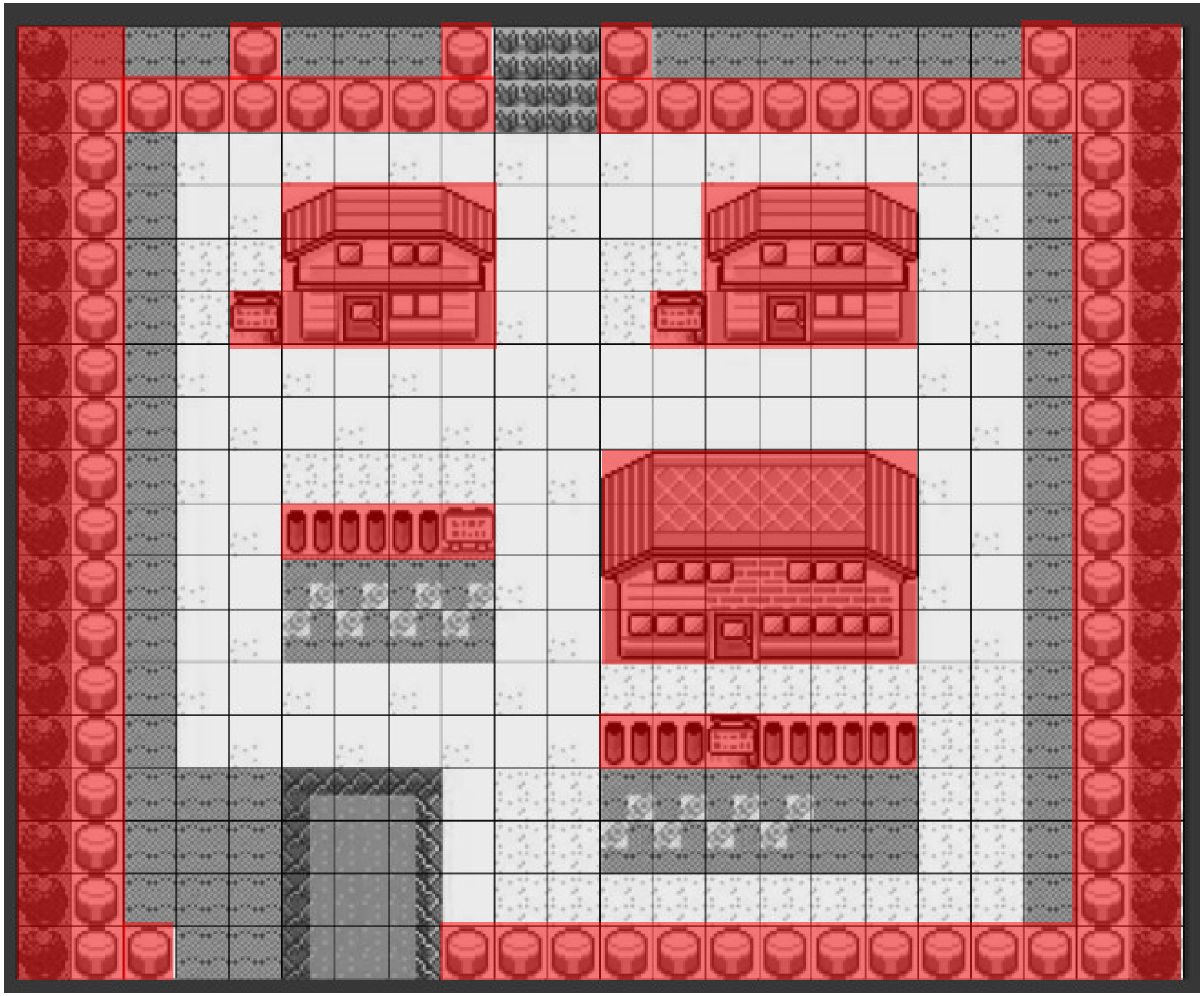


http://fightingkitten.webcindario.com/?p=8

Environment abstraction
Regular tiles

Regular tiles

- We mark some tiles as non-walkable

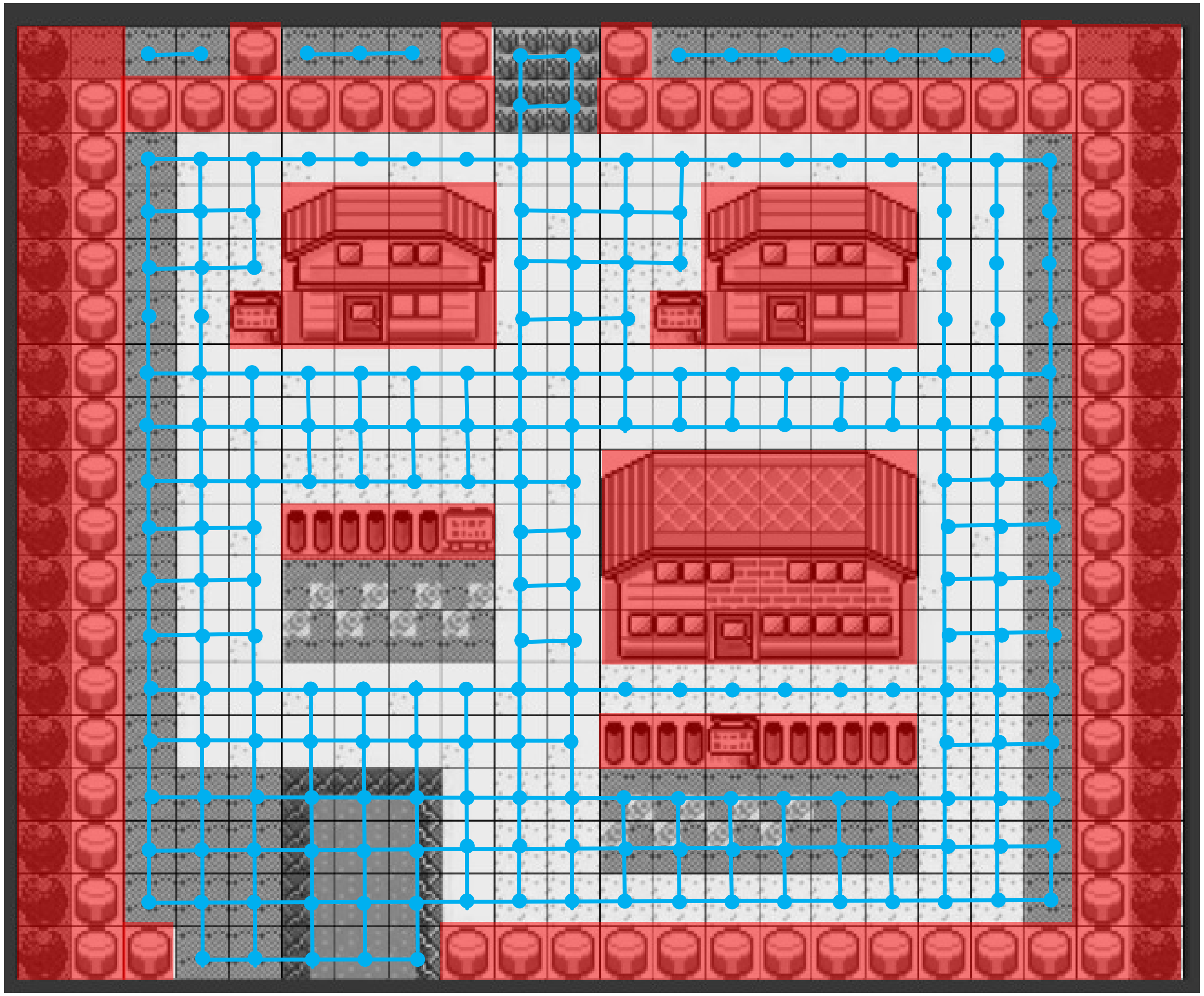


http://fightingkitten.webcindario.com/?p=8

Environment abstraction
Regular tiles

Regular tiles

- We mark some tiles as non-walkable
- And create a graph out of walkable tiles

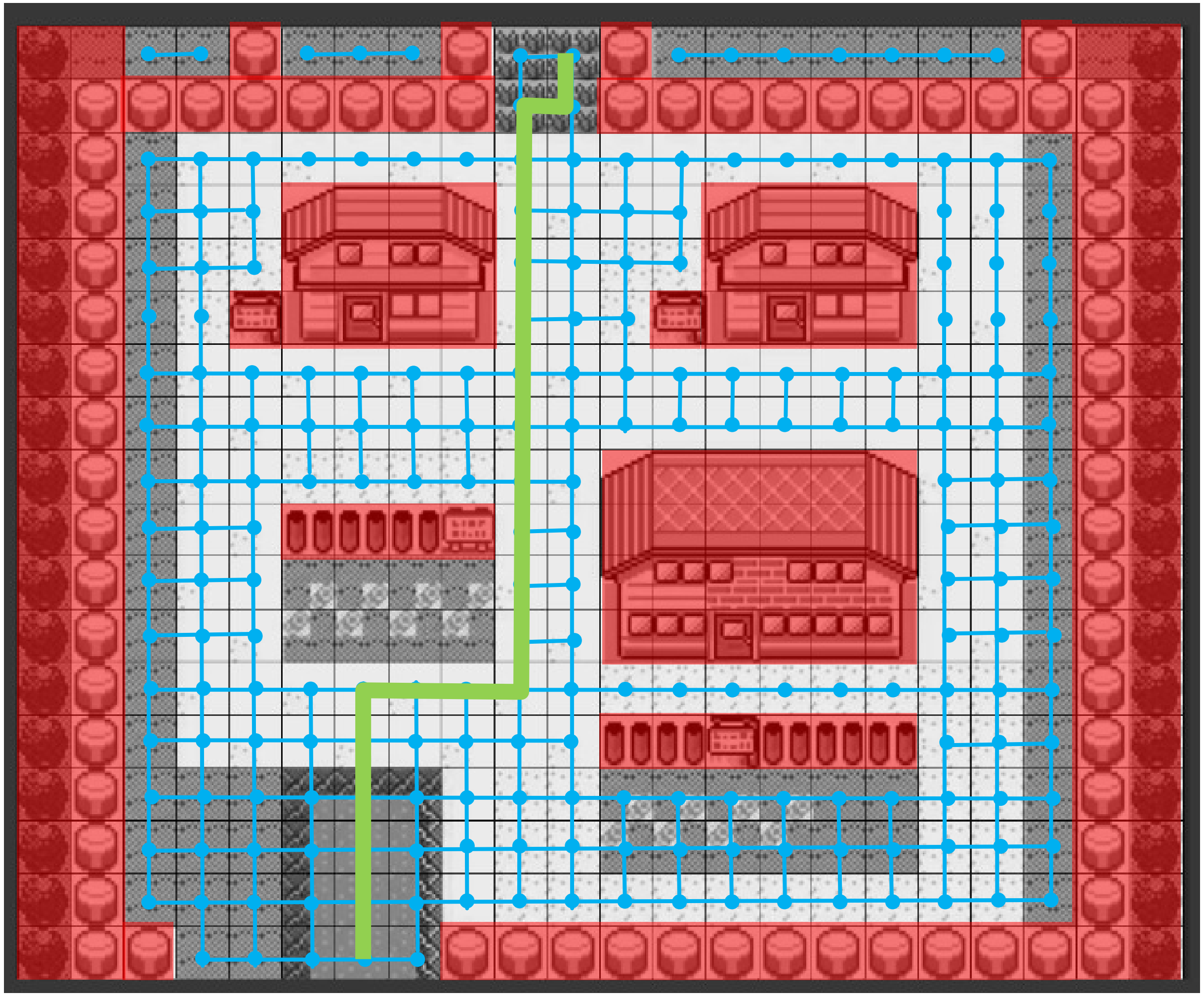


http://fightingkitten.webcindario.com/?p=8

Environment abstraction
Regular tiles

Regular tiles

- We mark some tiles as non-walkable
- And create a graph out of walkable tiles
- Which provides us with mechanical paths



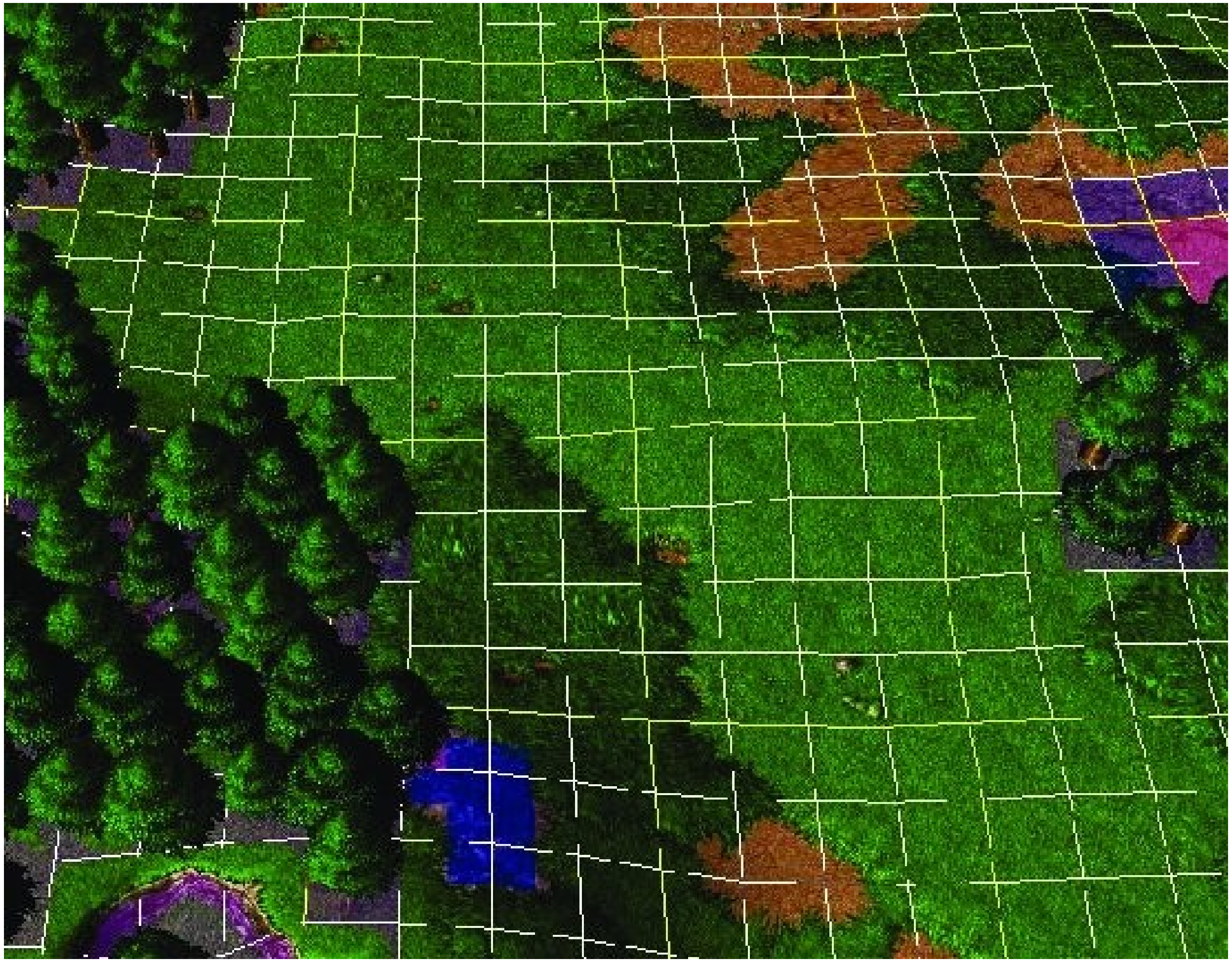
http://fightingkitten.webcindario.com/?p=8

Environment abstraction

Regular tiles

Dune II (1993)

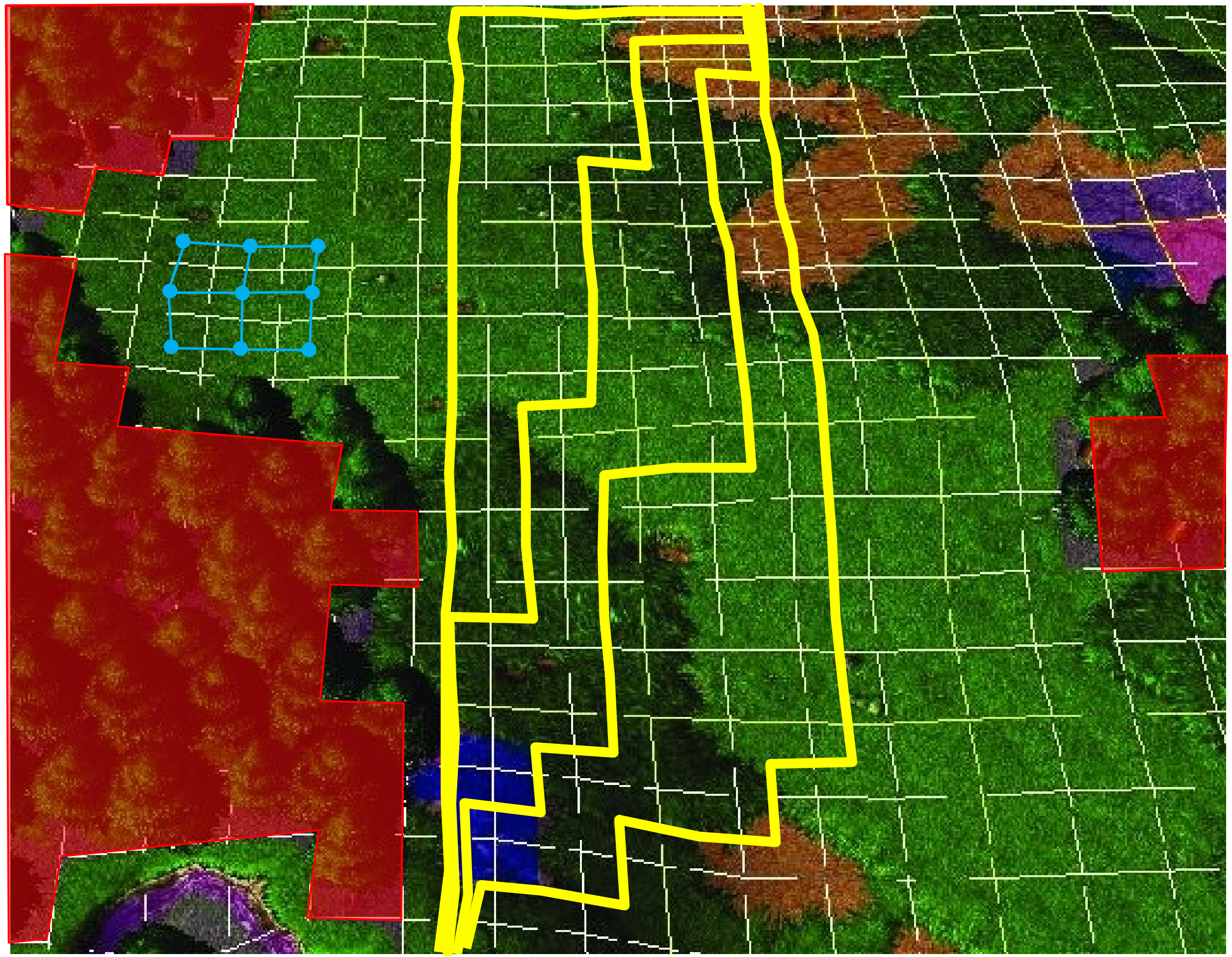
Warcraft II (1995)



Warcraft III, editor. Blizzard Entertainment (2002

Environment abstraction
Regular tiles

Regular tiles
- Can be used for terrain in 3D as well
- With the same trick (non-walkable, graph, paths)

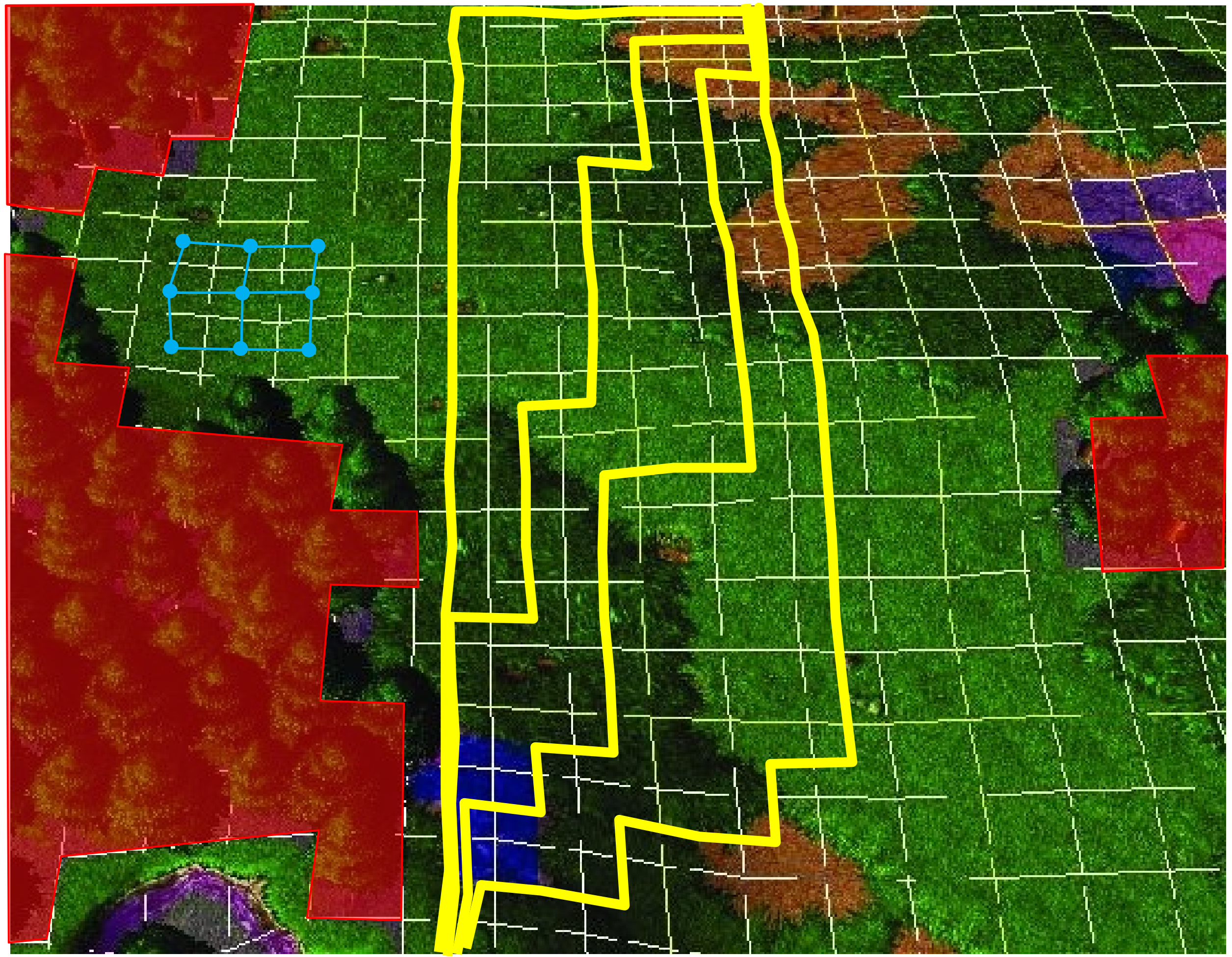


Warcraft III, editor. Blizzard Entertainment (2002

Environment abstraction
Regular tiles

Regular tiles

- Can be used for terrain in 3D as well
- With the same trick (non-walkable, graph, paths)
- Notice that pictured paths are of the same length! :-/

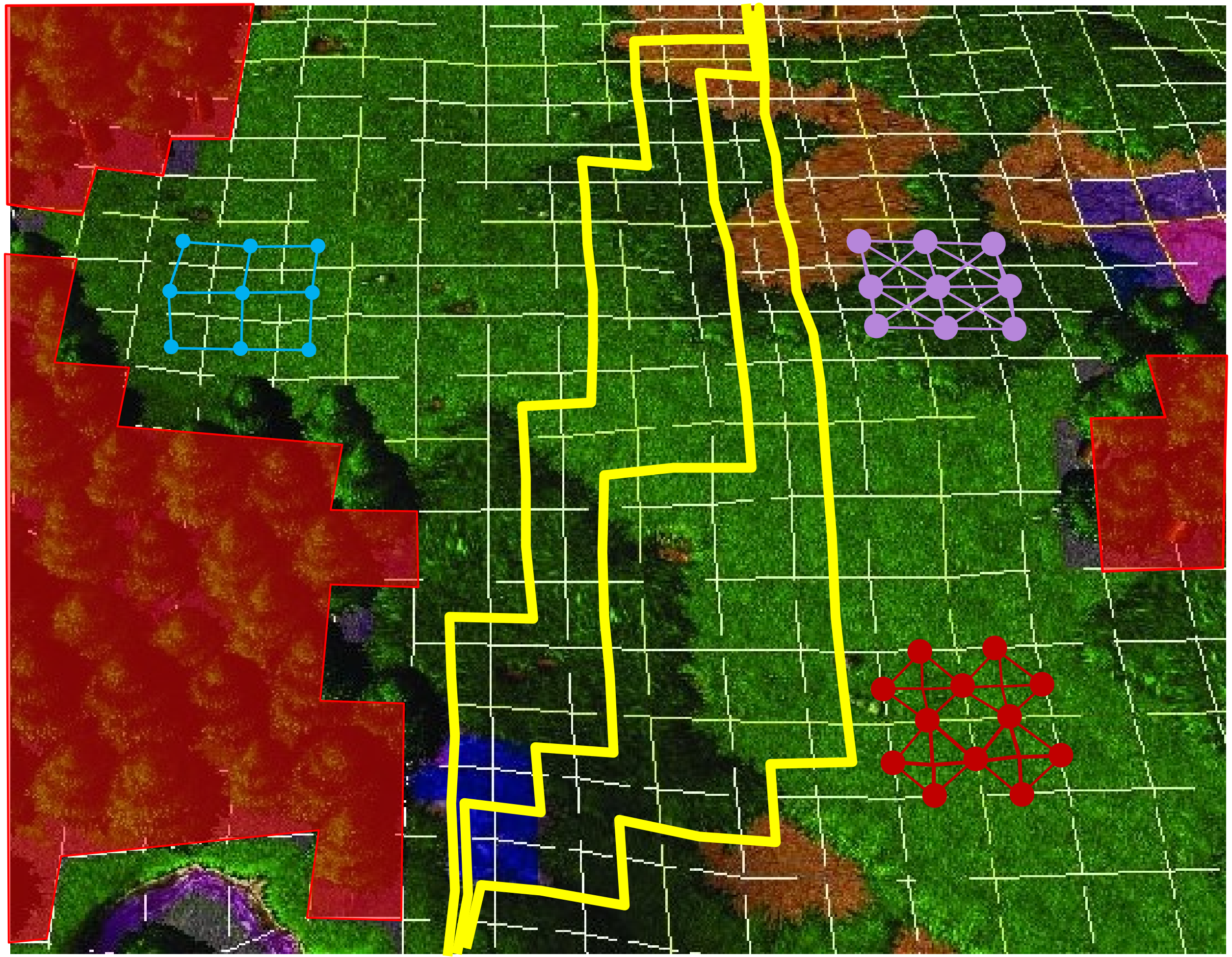


Warcraft III, editor. Blizzard Entertainment (2002

Environment abstraction
Regular tiles

Regular tiles

- Alternatively, we can create a different graph (twice as large)

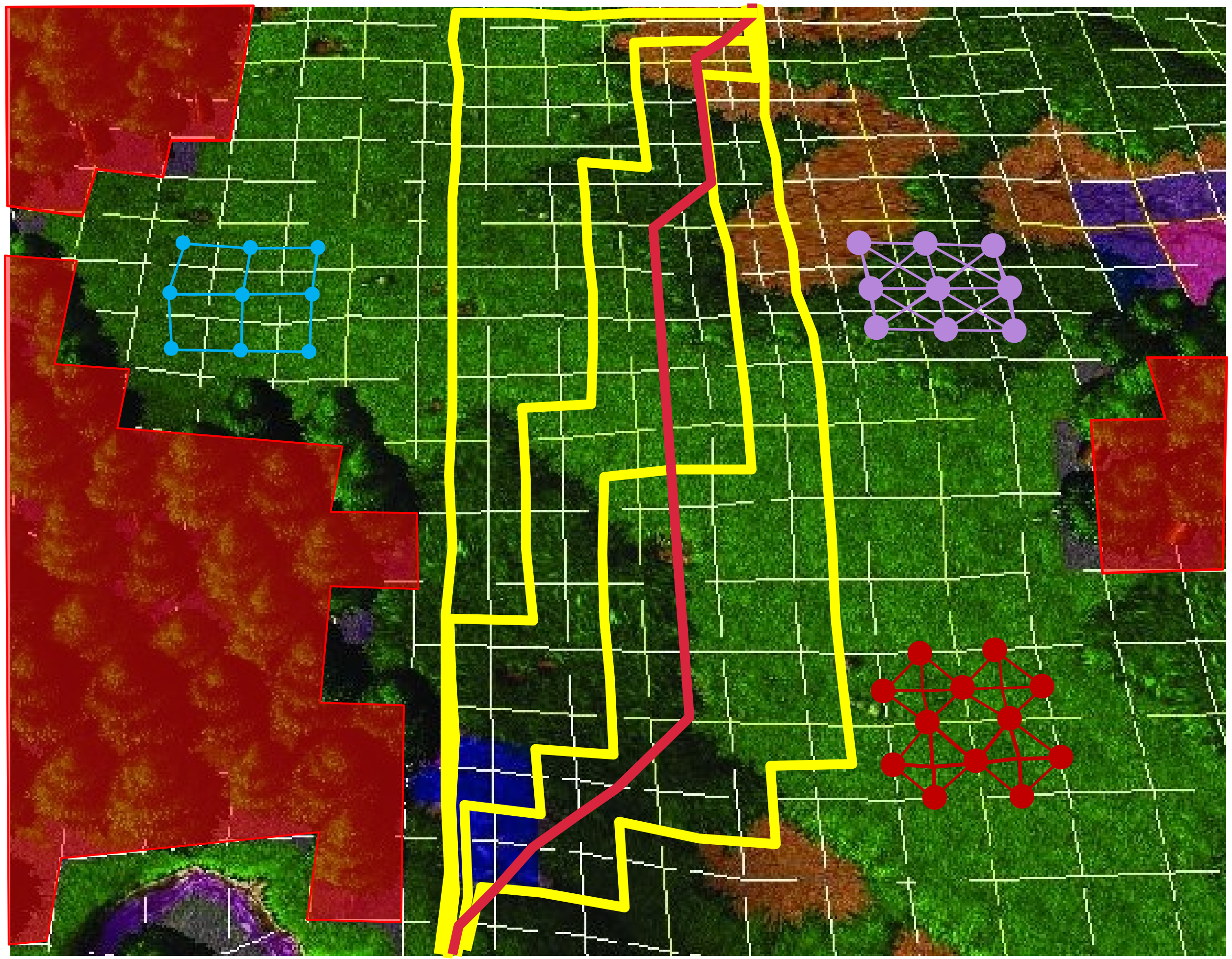


Warcraft III, editor. Blizzard Entertainment (2002

Environment abstraction
Regular tiles

Regular tiles

- Alternatively, we can create a different graph (twice as large)
- Which gives rise to more natural paths

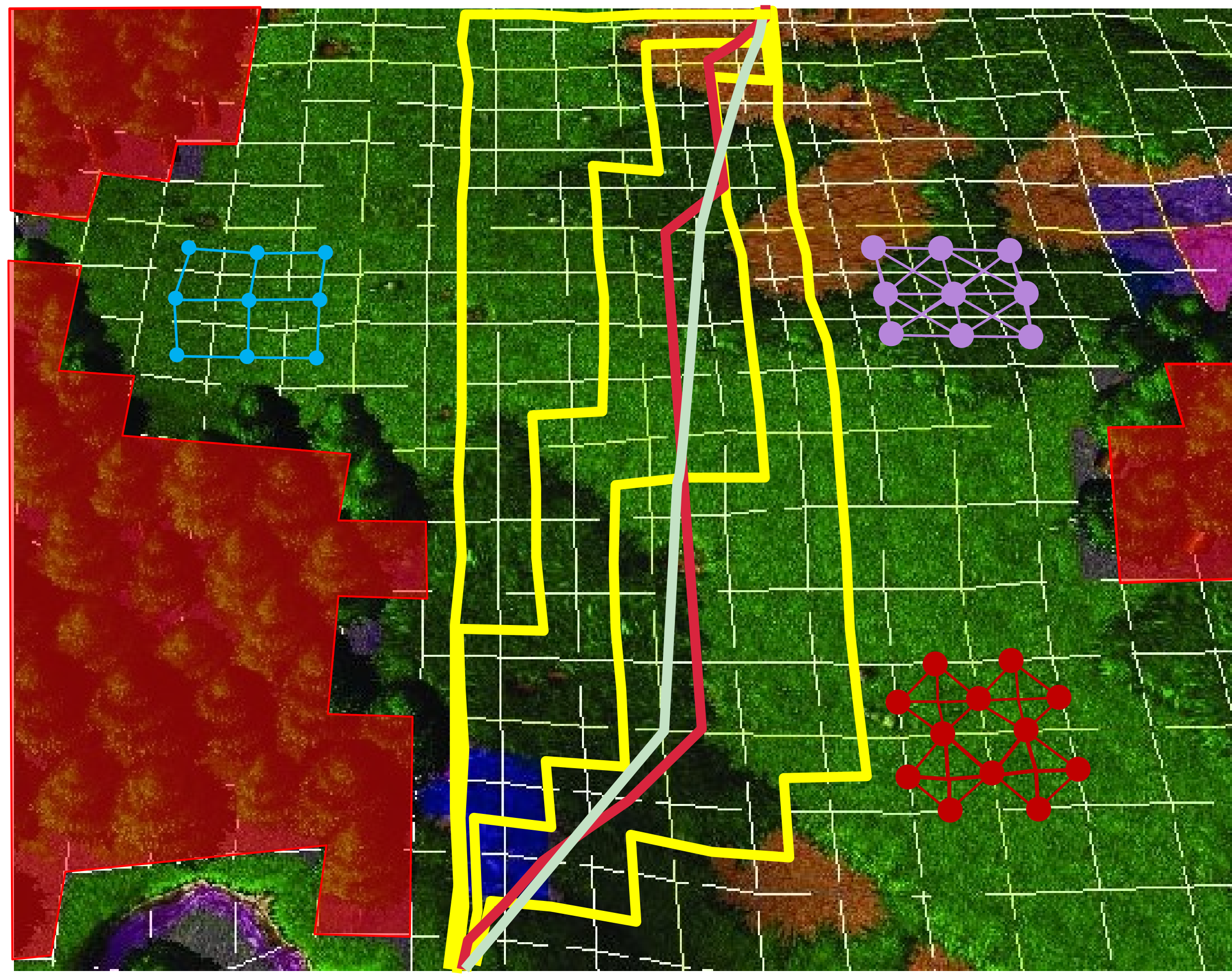


Warcraft III, editor. Blizzard Entertainment (2002

Environment abstraction
Regular tiles

Regular tiles

- Alternatively, we can create a different graph (twice as large)
- Which gives rise to more natural paths
- And we can smooth it with a funnel algorithm



Warcraft III, editor. Blizzard Entertainment (2002

Pathfinding
Environment Abstractions

Modelling terrain with a weighted graph

Pathfinding
Tiles and Costs

Terrain is often associated with a “travel cost”.

When turning tiles (or anything else in graph) we have two options how to model it

1. Associate the cost with nodes
2. Associate the cost with links (allows for different costs for traveling up/down the hill)

Pathfinding
The Algorithms

How to find a path?

Graph search algorithms

(the skeleton)

Graph search algorithms
The Skeleton (unidirectional)

Idea
- Start from origin
- Incrementally push newly touched nodes into the open-list (the frontier or the fringe)
- Select next nodes from the open-list according to a strategy

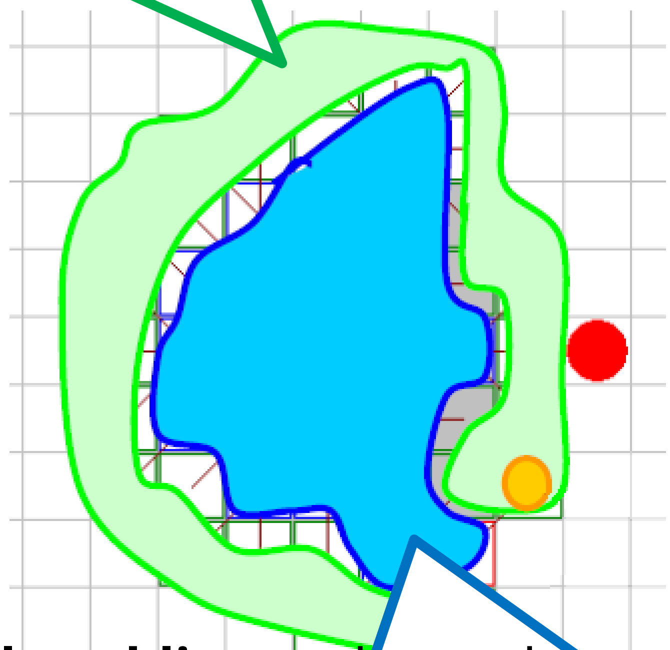
Plug-ins
- `Open-list` implementation
- `Strategy` of extraction
- (`Closed-list` implementation)

Algorithm template

```
1.  make open-list
2.  push start into open-list
3.  while open-list not empty
4.    extract node from open-
        list according to
        “strategy”
5.    if node is target
6.      return path to node
7.    else
8.      expand node by
          checking its direct
          neighbors possibly
          adding them
          to open-list
9.      move expanded node to
          closed-list
```

...gorithms
...ectional) - Vocabulary

The **open-list**, also called **fringe** or **frontier**, contains nodes that are currently considered for expansion, i.e., next nodes to check.

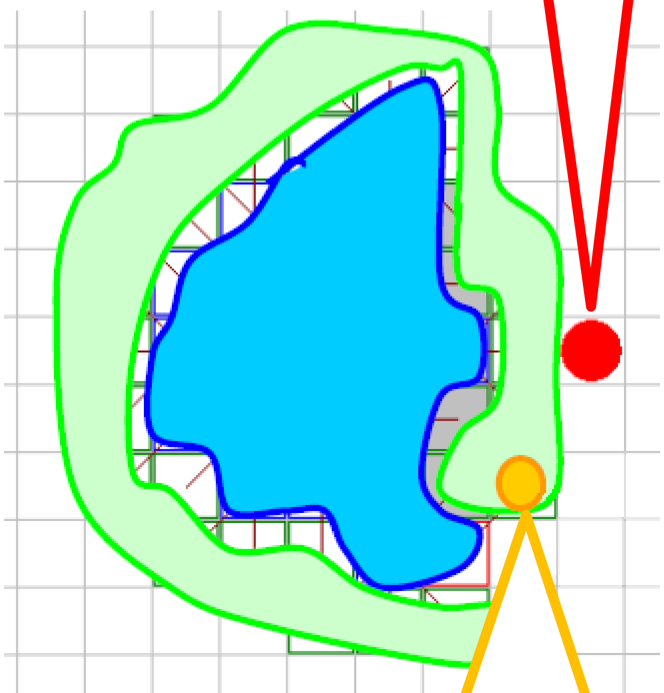


Closed-list, nodes we have already expanded; if the strategy is optimal, those nodes will not need to be moved back to the open-list (referred to as reopened).

Algorithm template

```
1. make open-list
2. push start into open-list
3. while open-list not empty
4.    extract node from open-
        list according to
        "strategy"
5.    if node is target
6.      return path to node
7.    else
8.      expand node by
          checking its direct
          neighbors possibly
          adding them
          to open-list
9.      move expanded node to
          closed-list
```

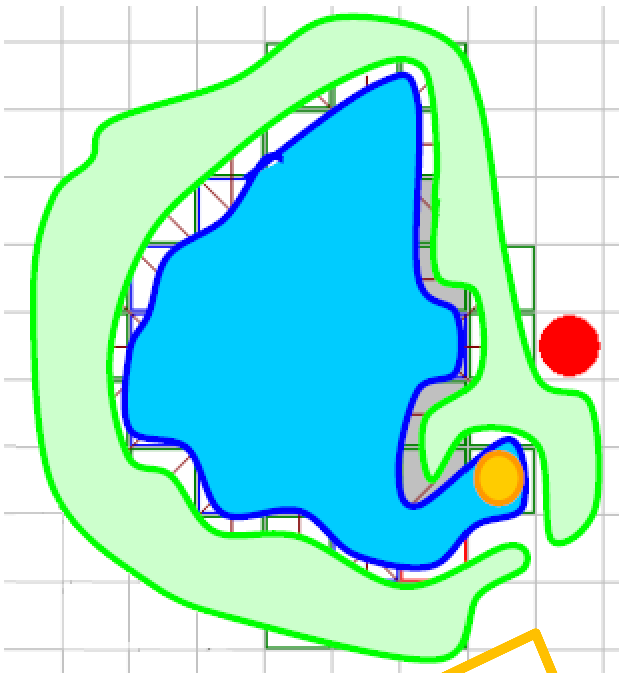
Graph se… …lgorithms
The Skelet… …rectional) - Vocabulary



Algorithm template

```
1.  make open-list
2.  push start into open-list
3.  while open-list not empty
4.      extract node from open-
          list according to
          "strategy"
5.      if node is target
6.        return path to node
7.      else
8.        expand node by
            checking its direct
            neighbors possibly
            adding them
            to open-list
9.        move expanded node to
            closed-list
10. return no-path
```

Graph search algorithms
The Skeleton (unidirectional) - Vocabulary



Expanded node was moved to the closed-list and it's neighbors, which were not in either lists, were moved into the open-list.

Algorithm template

```
1.  make open-list
2.  push start into open-list
3.  while open-list not empty
4.     extract node from open-
         list according to
         "strategy"
5.     if node is target
6.        return path to node
7.     else
8.        expand node by
            checking its direct
            neighbors possibly
            adding them
            to open-list
9.        move expanded node to
            closed-list
10. return no-path
```

Graph search algorithms
The Skeleton (unidirectional)

Notes

- More complex `open-list` or `strategy` is, more time it requires to compute one step of the search

⇒ Trade-offs
    Path quality vs.
    Path optimality vs.
    Terrain vs.
    First move delay vs.
    Computation time vs.
    Precomputations

Algorithm template

```
1.  make open-list
2.  push start into open-list
3.  while open-list not empty
4.     extract node from open-
          list according to
          “strategy”
5.     if node is target
6.        return path to node
7.     else
8.        expand node by
             checking its direct
             neighbors possibly
             adding them
             to open-list
9.        move expanded node to
             closed-list
10. return no-path
```

Pathfinding
The Algorithms

How to find the path?

Graph search algorithms

(instances)

Graph search algorithms
Breadth-first search (BFS)

Open-list
A queue (first-in first-out)

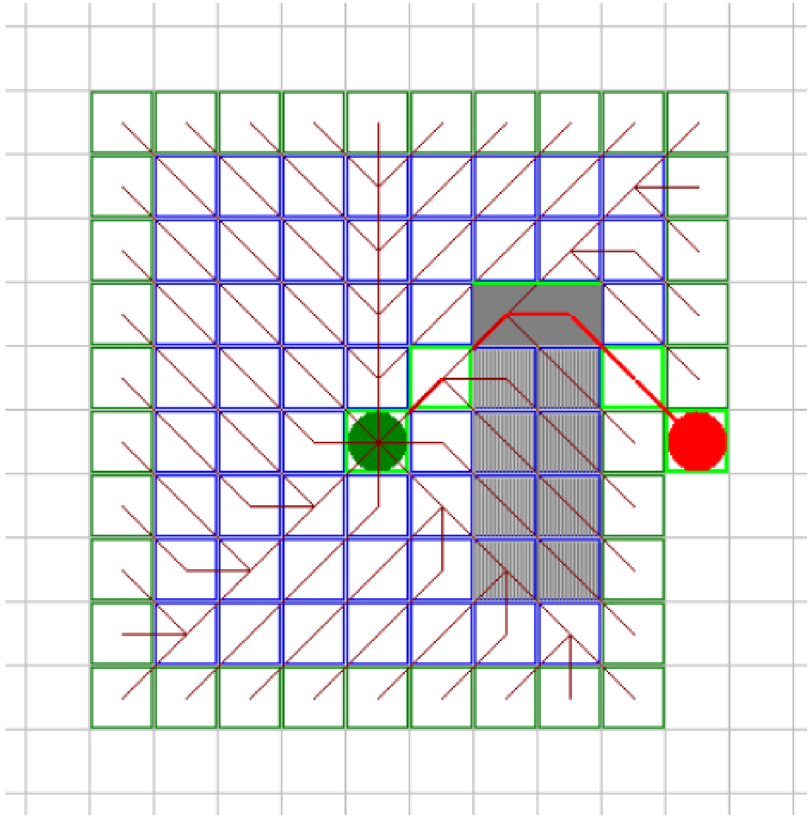
Strategy
To select the first node in the queue

Notes

? Expands nodes uniformly around
  the origin without checking
  its terrain cost

\- Expands a lot of nodes

⇒ Good for “checking what is around”



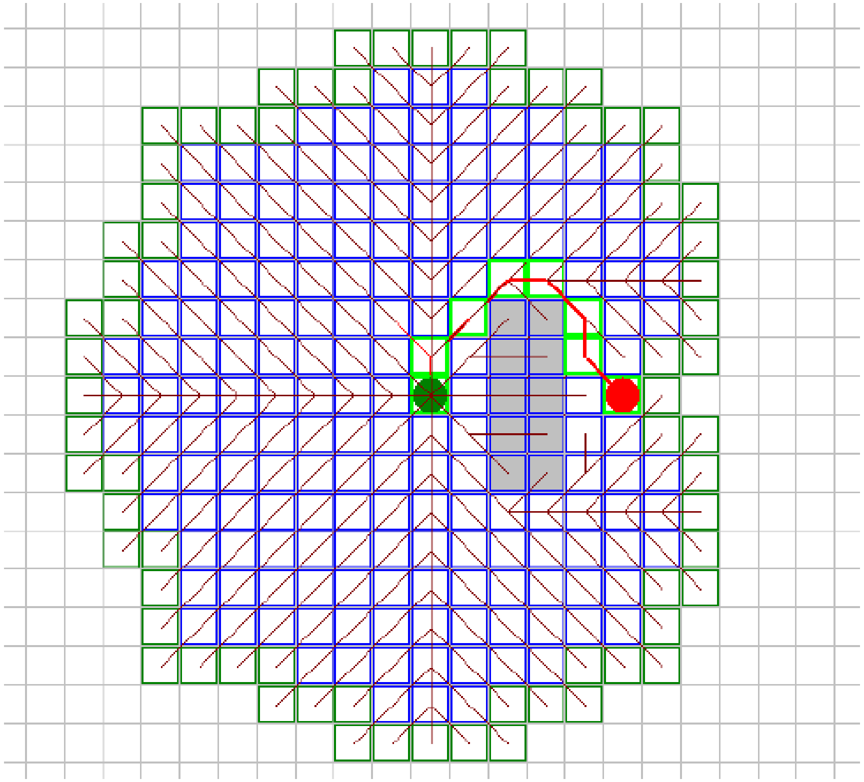
Graph search algorithms
Dijkstra's algorithm

`Open-list`
A prioritized queue (first-in first-out); nodes are sorted according to their path-distance from the start

`Strategy`
To select the first node from the queue

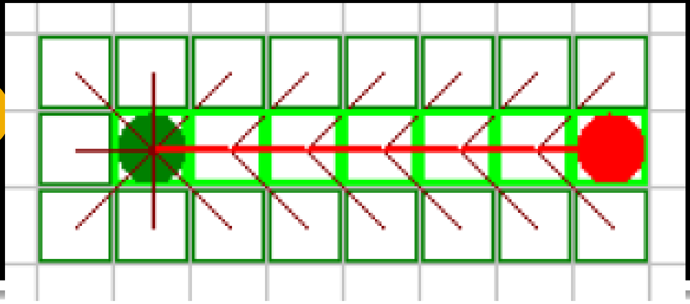
Notes
+ Takes terrain into account
+ Does not assume anything about the topology of the environment (works with, e.g., teleports)
- Expands a lot of nodes

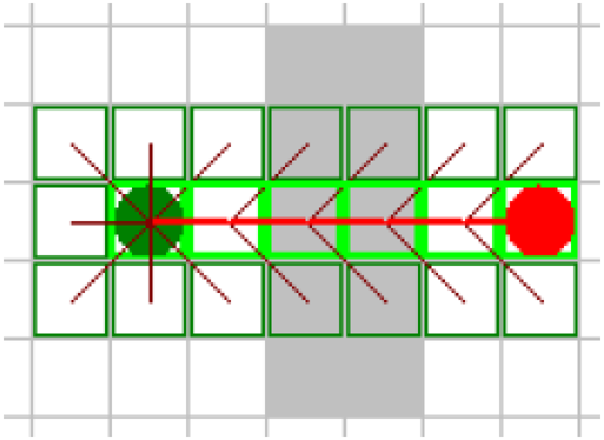
⇒ Good for "checking what is around in certain path-distance"

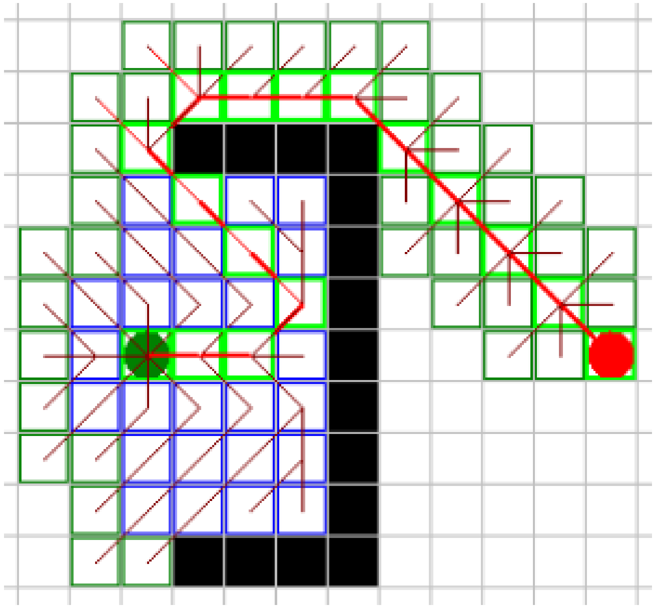


Graph search algo

Best-first search







Open-list

A prioritized queue; nodes sorted according to the estimated cost to the target

Strategy

To select the node that seems to be the closest to target (according to heuristic function), i.e., the first node from the queue

Notes

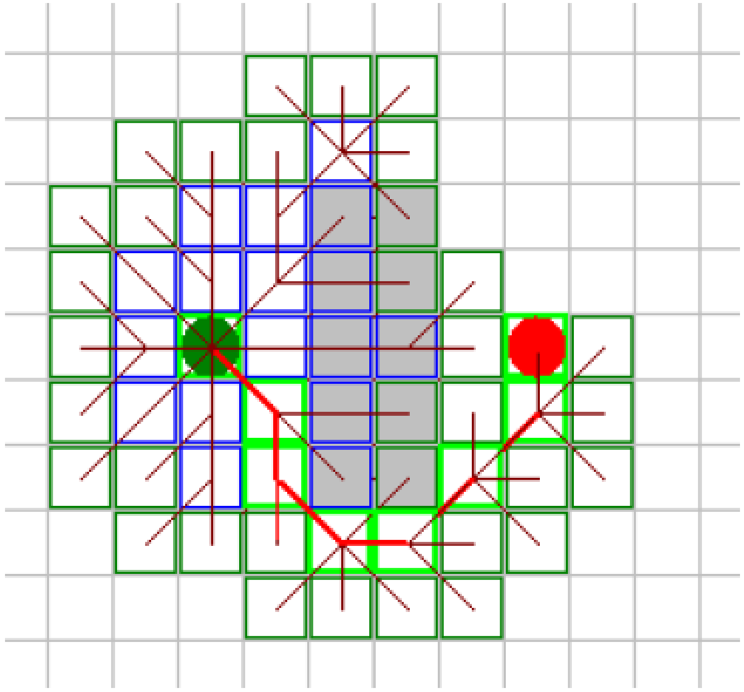
- ! Requires a heuristic function h: $node \rightarrow R$
- \+ Computes fast in open-space
- \- Does not guarantee the shortest path
- \- Does not take terrain into account

Graph search algorithms
A* search

Open-list
A prioritized queue; nodes sorted according to the (path cost from start + estimated cost to target)

Strategy
To select the node that seems to be the most promising, i.e., the first node from the queue

Notes
! Requires a heuristic function h: $node \rightarrow R$
+ Computes faster than Dijkstra
+ Guarantee the shortest path if heuristic is done right
+ Takes terrain into account



Graph search algorithms
A* search

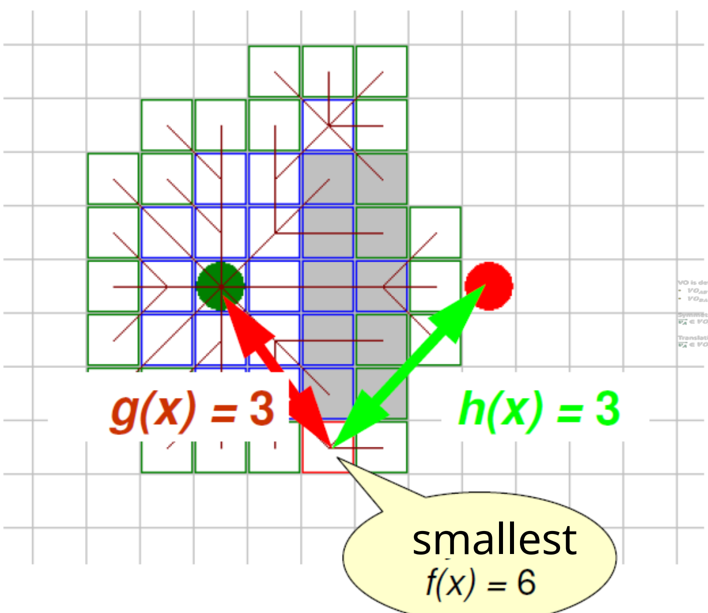
`Open-list`

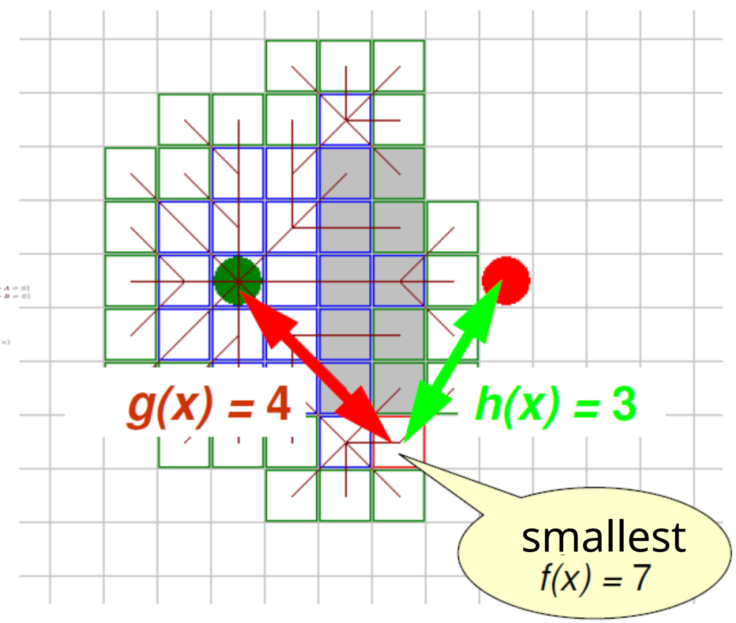
A prioritized queue; nodes sorted according to the:

$$f(x) = g(x) + h(x)$$

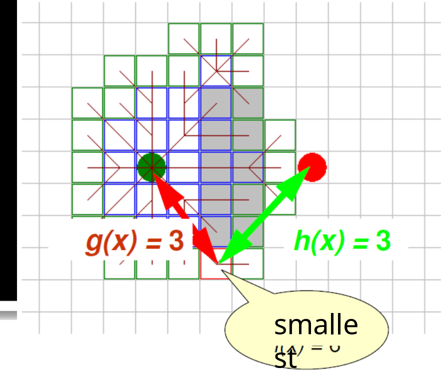
$g(x)$ … path cost from start

$h(x)$ … estimated cost to target the heuristic function





A* Search
Properties



Open-list
A prioritized queue; nodes sorted according to the:

$$f(x) = g(x) + h(x)$$

$g(x)$ ... path cost from start
$h(x)$ ... estimated cost to target
the heuristic function

Def.: If $h(x) \leq$ real-path-cost to target, $h(x)$ is called **admissible**.

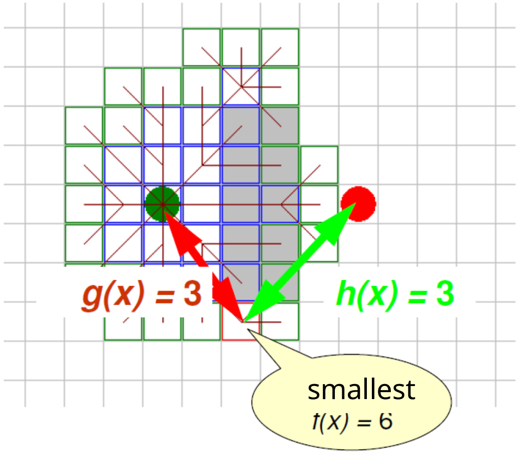
Theorem: A* admissibility
If graph costs are non-negative and $h(x)$ is **admissible**, A* finds an optimal (the cost-shortest) path in the finite number of steps if it exists or terminates.

Def.: If for every edge $(a \rightarrow b)$: $h(a) \leq cost(a \rightarrow b) + h(b)$,
$h(x)$ is called **monotone** or **consistent**.

Theorem: A* optimality
If graph costs are non-negative and $h(x)$ is **consistent**, A* does not reopen nodes in the closed list.

Def.: $h(x)$ that is both admissible and consistent is called **feasible**.

A* Search
Heuristic functions



- Defined as $h(x)$, but in reality implemented as $h(x,t)$, where t is target node we are finding the path to
- Depending on the environment and move actions we typically use following feasible functions

  Regular grid, 4-way cells: $h(x,t) = |x_x - t_x| + |x_y - t_y|$

  Regular grid, 8-way cells: $h(x,t) = max(|x_x - t_x|, |x_y - t_y|)$

  $$\text{2D: } h(x,t) = \sqrt{(x_x - t_x)^2 + (x_y - t_y)^2}$$

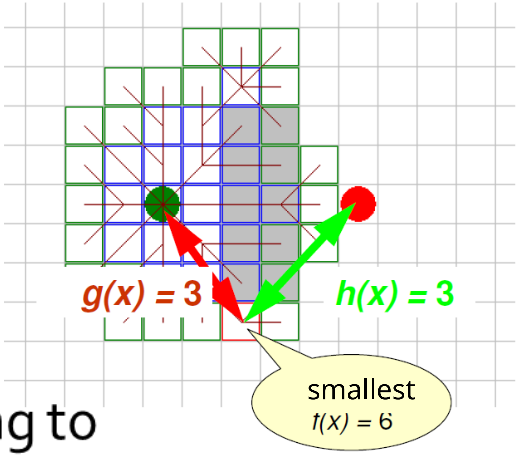
  $$\text{3D: } h(x,t) = \sqrt{(x_x - t_x)^2 + (x_y - t_y)^2 + (x_z - t_z)^2}$$

- Cannot be used if there are teleports on the map!
- For teleports $r_1(a_1 \to b_1), \ldots, r_n(a_n \to b_n)$ on the map, we need following modification

  $$h'(x,t) = \min\{h(x,t), \min_{i=1..n}\left(h(x, r_i.a_i) + h(r_i.b_i, t)\right)\}$$

  - Quite costly though! And not complete as it assumes use of single teleport

A* Search
Relation to other searches

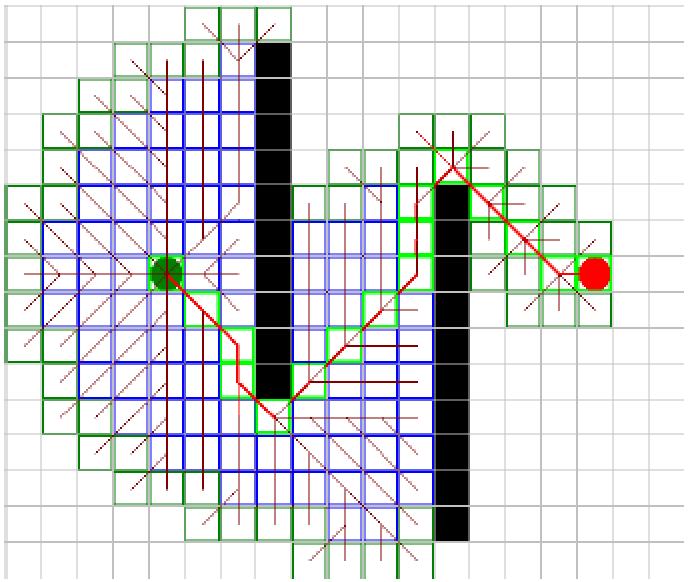


- A* strategy is ordering nodes in open-list according to $f(x, s, t) = g(s, x) + h(x, t)$
  - x … node in open-list; s … start node; t … target node
  - $g(s, x)$ … current shortest-path known between s and x
  - $h(x, t)$ … estimation of the path cost between x and t
  - More on the properties of f and strategies for combining g a h: Dechter, R., & Pearl, J. (1985). Generalized best-first search strategies and the optimality of A*. *Journal of the ACM (JACM)*, *32*(3), 505-536.

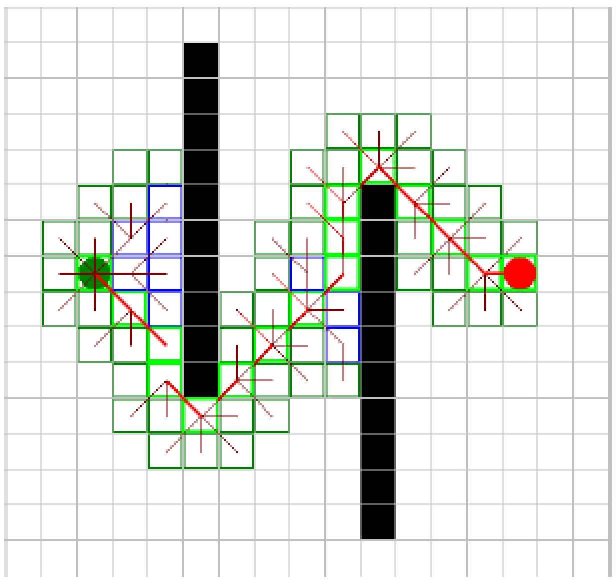
- Simulating other algorithms
  - BFS: $g(s, x) = least\ \#edges\ between\ s\ and\ x$, $h(x, t) = 0$
  - Dijkstra: $h(x, t) = 0$
  - Best-first: $g(s, x) = 0$

A* Search
Optimization

- When A* hits obstacle, it expands a lot of unnecessary nodes
- We can use non-admissible heuristic to force A* to expand the promising nodes faster => does not always return the optimal path then
- Can be used in open environments, elsewhere value between 1.5 and 2.0 is typically used

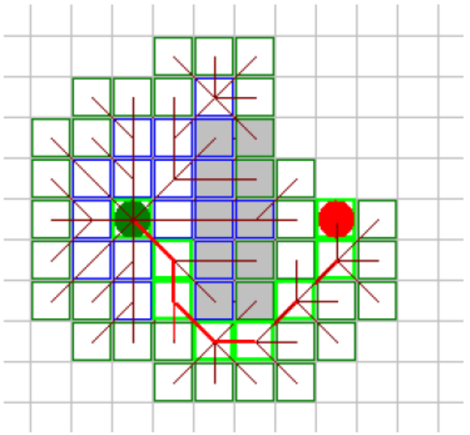


$f(x) = g(x) + h(x)$



$f(x) = g(x) + 5.h(x)$

A* Search
More notes



- Requires a heuristic function
  $h: node \rightarrow R$

+ Computes faster than Dijkstra; optimal path found if exists and the heuristic is admissible
+ Takes terrain into account
+ Easy to implement
+ Can be tweaked

- If path does not exist, searches through the whole space
- “Big” first move delay (search has to finish first)
- Does not cope with dynamic environment
- Works only for individuals, does not take other agents into account

Actually the negatives holds for all graph search algorithms seen so far…

„AI research often focuses in a direction that is less useful for games. A is the most successful technique that AI research has come up with—and nearly the only one applied in computer games.“* (A. Nareyek, **2004**)

Pathfinding
The Algorithms

How to find a path?

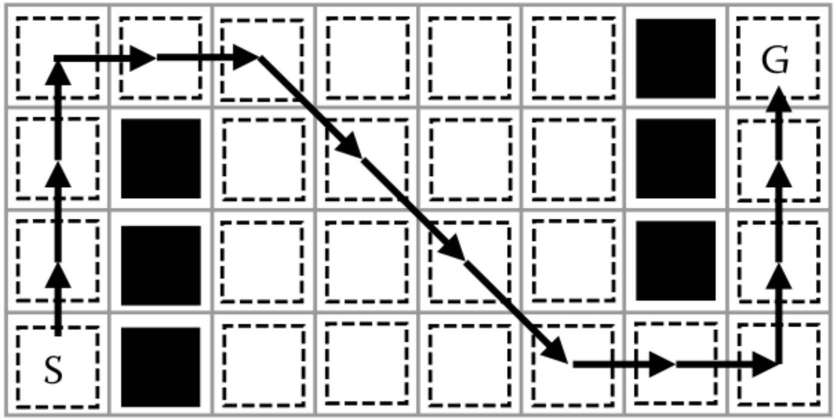
Speed-ups for regular grid

JPS

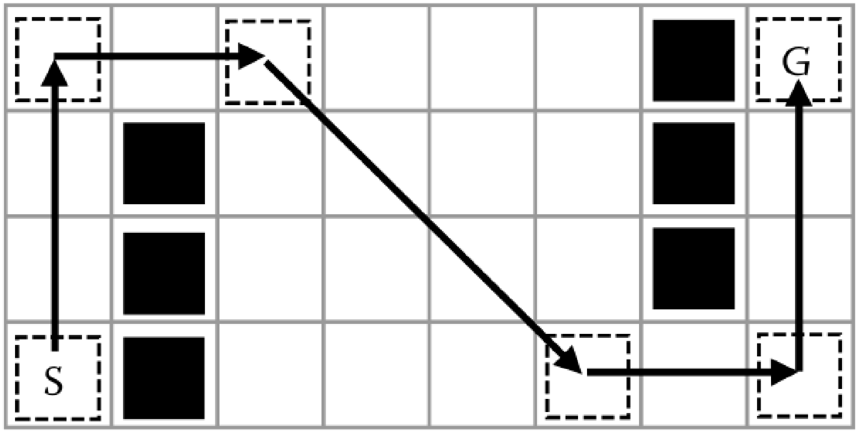
Harabor, D. D., & Grastien, A. (2011, August). Online graph pruning for pathfinding on grid maps. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*.

Improved A* on regular grids

- A* expands lots of nodes on grid-based maps
- Jump Point Search (JPS) performs “jump” lookaheads improving A* running time by about 10x



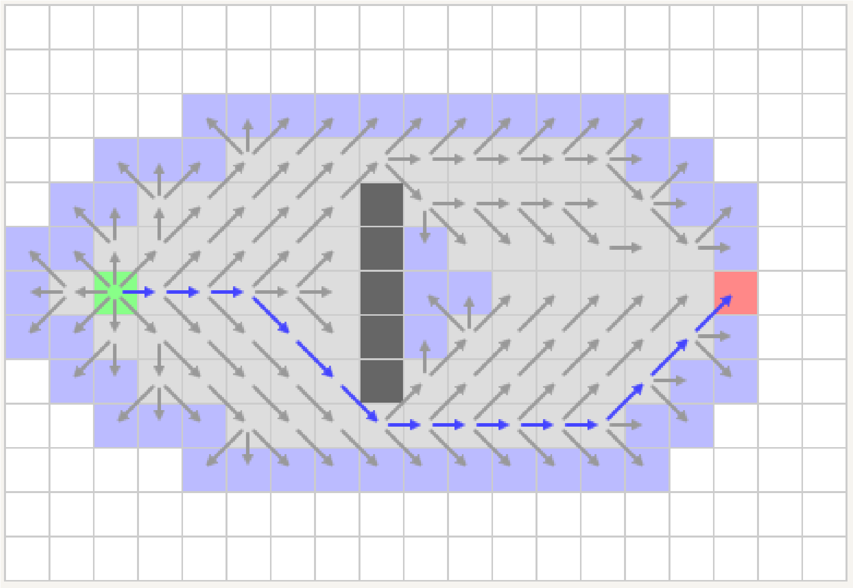
Traditional A*



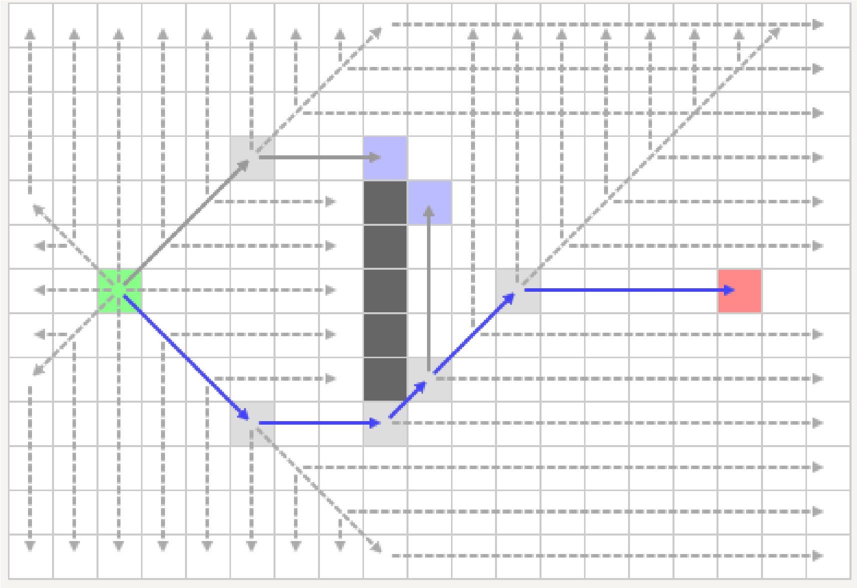
A* using JPS

JPS
Improved A* on regular grids

- Another example



Traditional A*

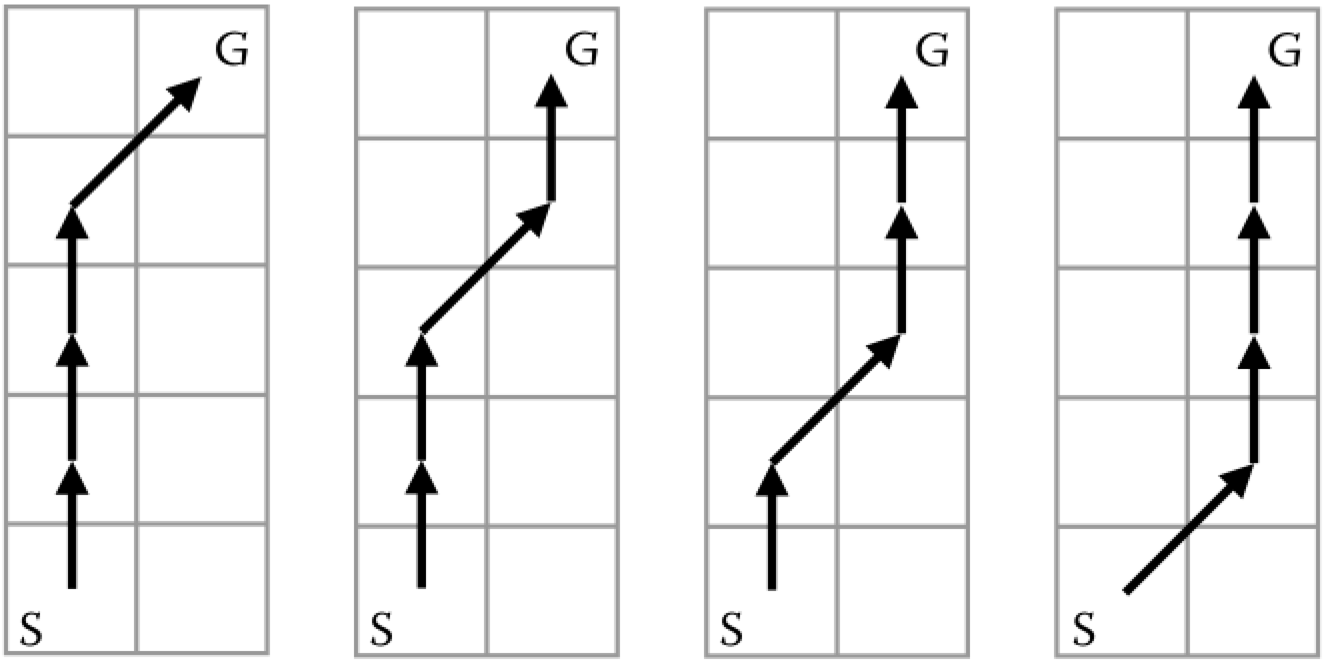


A* using JPS

Images from: https://zerowidth.com/2013/a-visual-explanation-of-jump-point-search.html

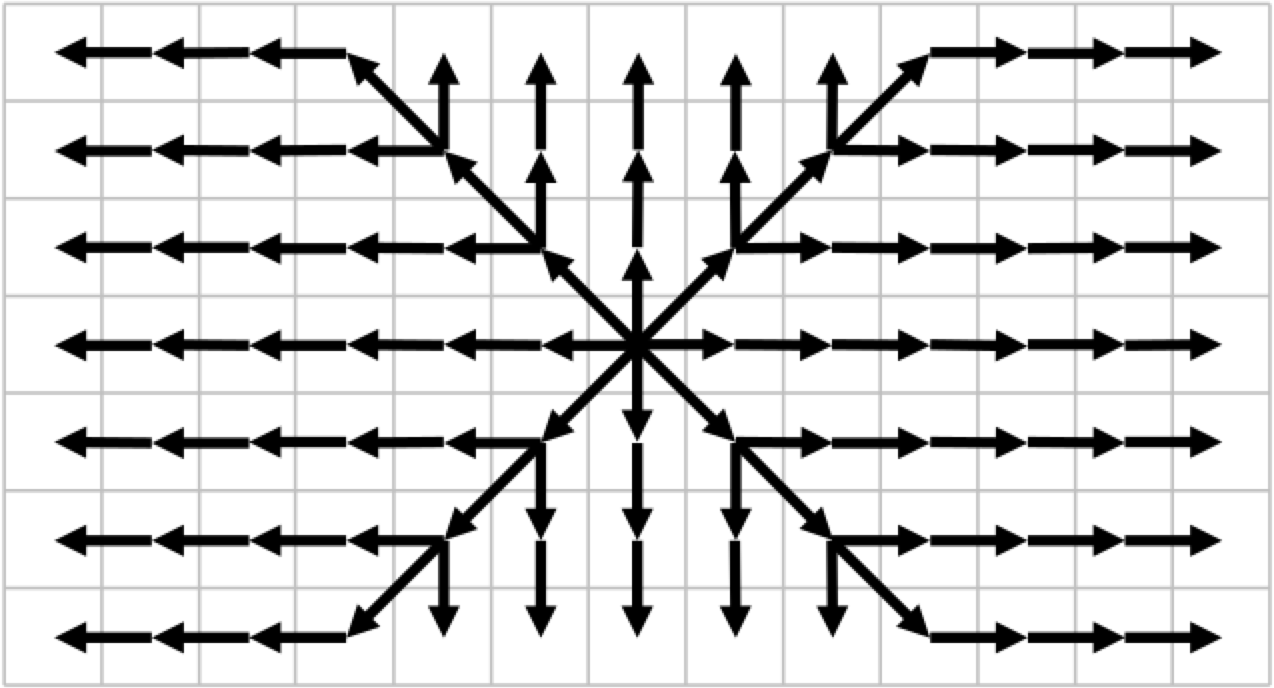
JPS
Improved A* on regular grids

- Works on an undirected uniform-cost grid
- Straight moves cost 1, diagonal moves cost sqrt(2)
- Traditional A* will check many paths with equivalent cost!



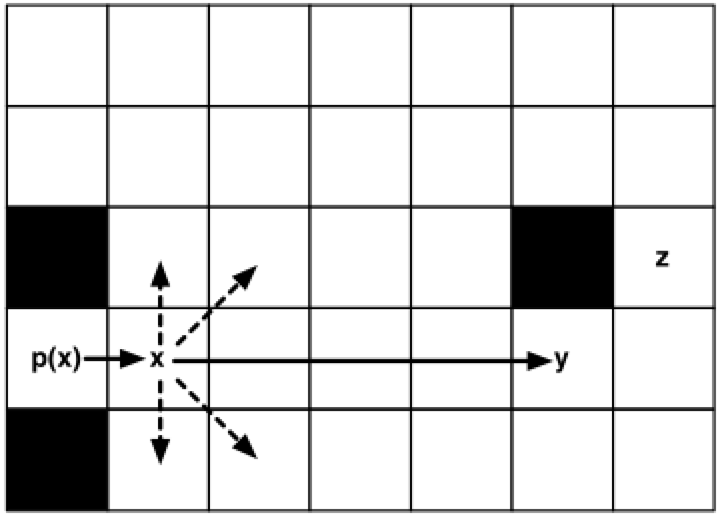
JPS
Improved A* on regular grids

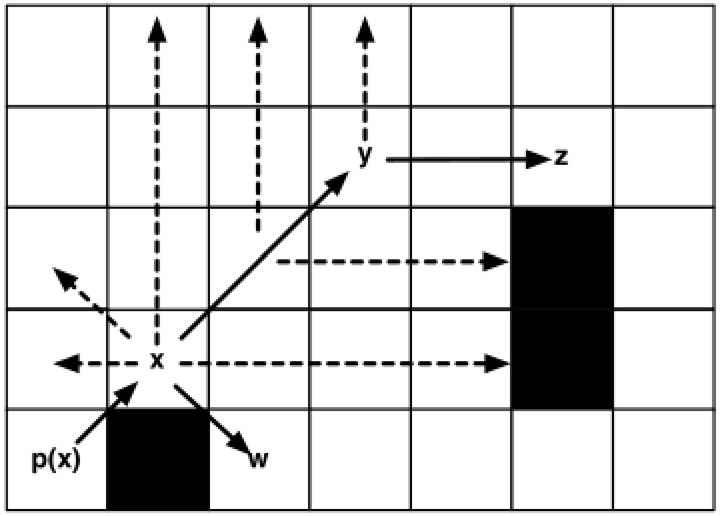
- JPS prunes most successor nodes
- In open space, only searches in directions seen here



JPS
Improved A* on regular grids

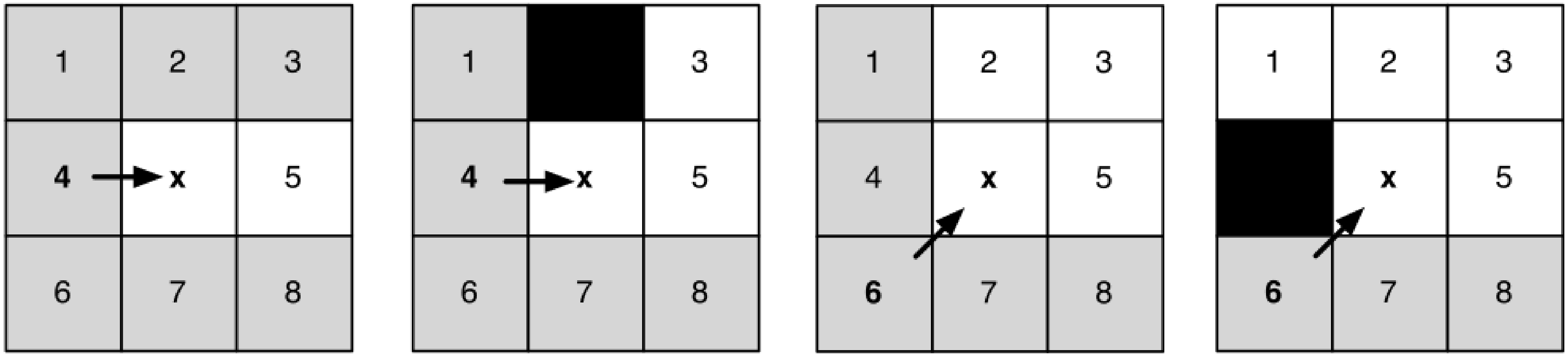
- A jump point is an "important" node we must add to the open list
- Below, dashed lines do not yield any jump point successors





JPS
Pruning rules

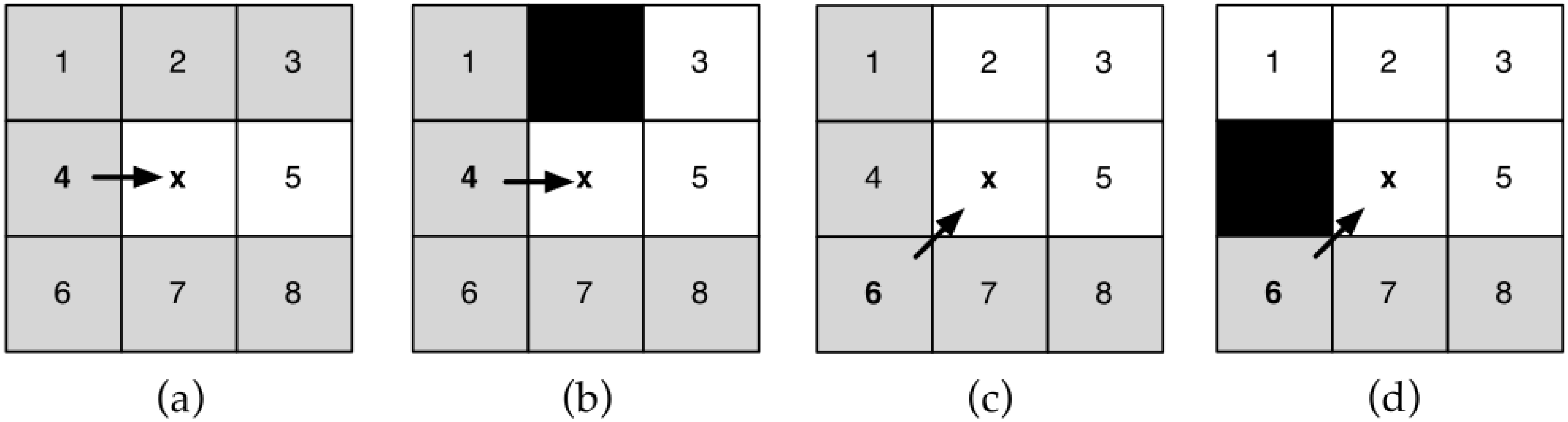
- Straight moves: prune neighbors that could be reached from the parent by another path of **shorter or equal** length
- Diagonal moves: prune neighbors that could be reached from the parent by another path of **strictly shorter** length



(a) (b) (c) (d)

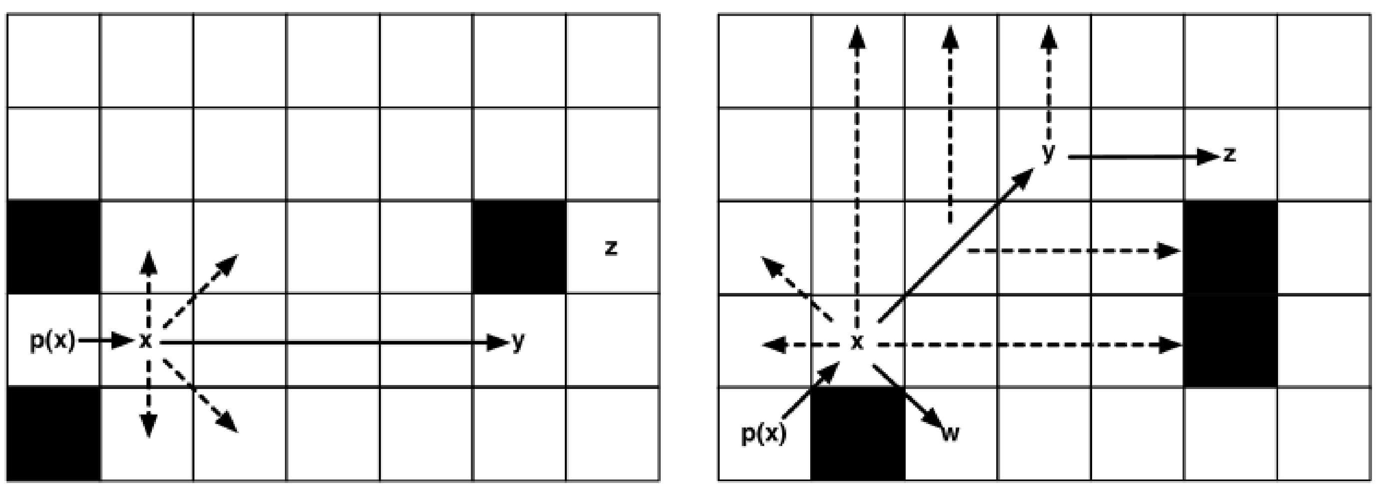
JPS
Forced and natural neighbors

- When there are no obstacles nearby, all neighbors are *natural*
- Obstacles produce additional *forced* neighbors



JPS
Jump point: definition

- Node y is the *jump point* from node x, heading in direction $\vec{d}$, if y minimizes k such that $y = x + k\vec{d}$ and one of:
    1. y is the goal node
    2. y has at least one forced neighbor
    3. $\vec{d}$ is a diagonal move and there exists $z = y + k_i\vec{d_i}$ where $\vec{d_i} \in \{\vec{d_1}, \vec{d_2}\}$ and z is a jump point from y by condition 1 or 2
- $\{\vec{d_1}, \vec{d_2}\}$ are the straight directions at 45 degrees to d

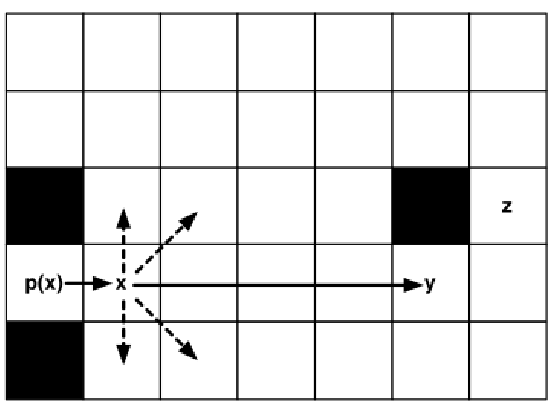


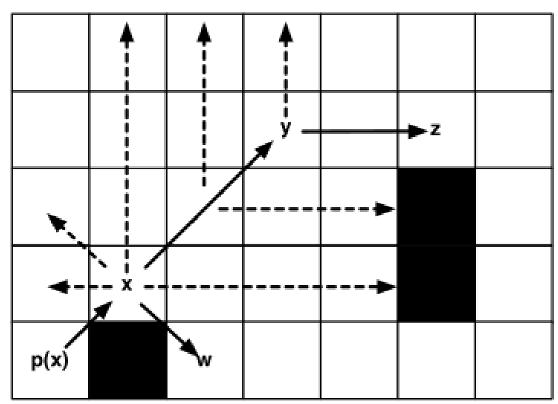
JPS
Algorithm

Algorithm 1 Identify Successors

Require: x: current node, s: start, g: goal

$successors(x) \leftarrow \emptyset$
$neighbours(x) \leftarrow prune(x, neighbours(x))$
for all $n \in neighbours(x)$ **do**
    $n \leftarrow jump(x, direction(x, n), s, g)$
    add n to $successors(x)$
end for
return $successors(x)$





JPS Algorithm

Algorithm 2 Function *jump*

Require: x: initial node, $\vec{d}$: direction, s: start, g: goal

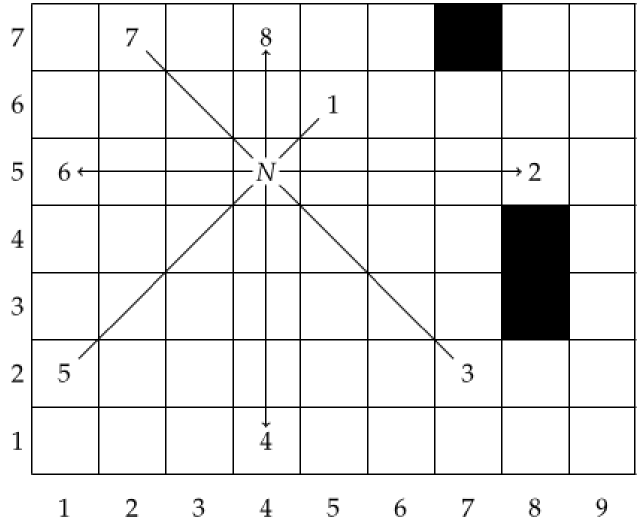
$n \leftarrow step(x, \vec{d})$
if n is an obstacle or is outside the grid **then**
    return *null*
end if
if $n = g$ **then**
    return n
end if
if $\exists\ n' \in neighbours(n)$ s.t. n' is forced **then**
    return n
end if
if $\vec{d}$ is diagonal **then**
    for all $i \in \{1, 2\}$ **do**
        if $jump(n, \vec{d}_i, s, g)$ is not *null* **then**
            return n
        end if
    end for
end if
return $jump(n, \vec{d}, s, g)$

JPS+

Rabin, S., & Silva, F. (2015). An Extreme A* Speed Optimization for Static Uniform Cost Grids [J]. *Game AI Pro 2: Collected Wisdom of Game AI Professionals*, 131.

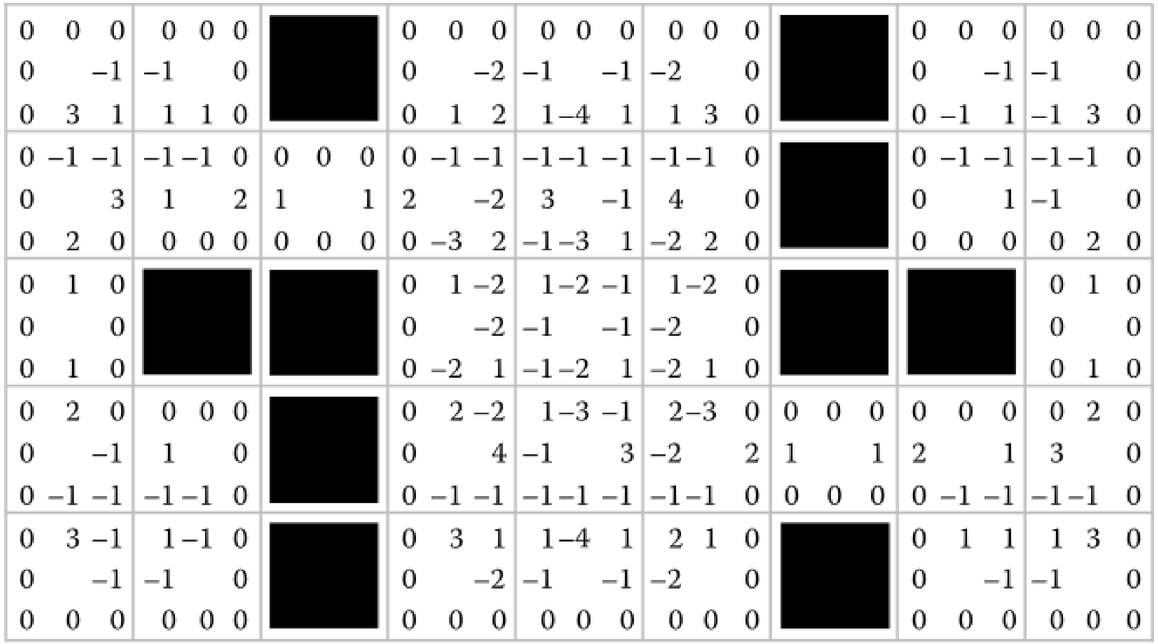
Baking JPS into the grid

- We can use precomputation to make JPS even faster!
- For each node, precompute jump points in all 8 directions
- Below, jump points 1-3 are ordinary
- Jump points 4-8 are *sterile* and would normally be discarded, but we keep them anyway



JPS+
Baking JPS into the grid

- For each direction, store the distance to the next jump point
  - or a value ≤ 0 for a distance to a sterile jump point
- A 16-bit integer is enough



JPS+
Finding the target node

- Precomputed jump points are independent of the target!
- At run time, check whether the path to each sterile successor crosses the row or column of the target T at some point J
  - If so, add a new jump point at J if there is a direct path from J to T