AI for video games

Fabrice Lamarche
ESIR

Schedule

- Motion planning
  - Deterministic methods
  - Potential fields
  - Probabilistic methods
- Search algorithms
  - Dijsktra, Wavefront, A*, IDA*
- Steering behaviors
  - Path following
  - Collision avoidance
  - Groups
- AI
  - Rule based systems
  - State machines
  - Hierarchical Task networks

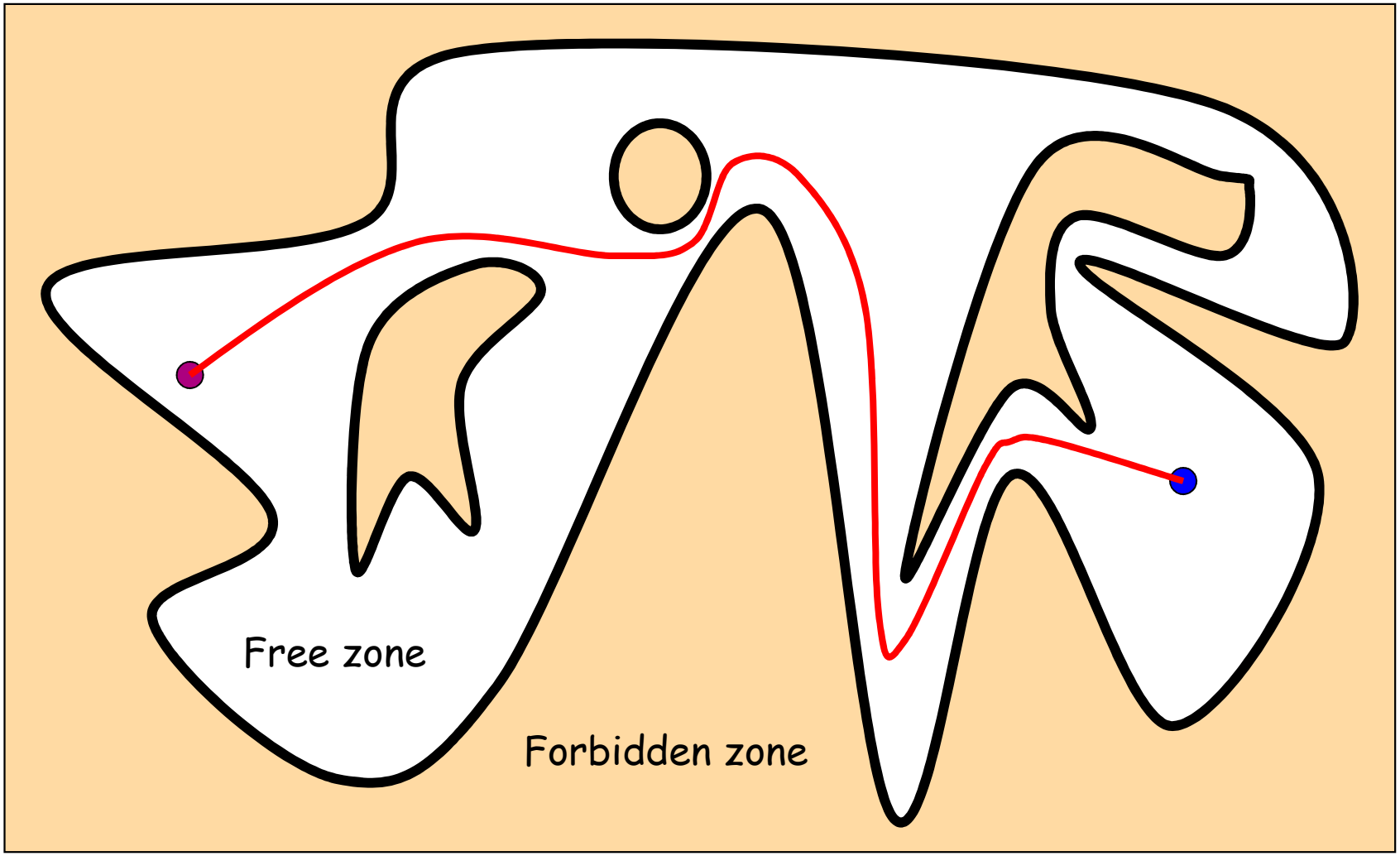
Some ressources

- http://www.aigamedev.com
- AI Game programming Wisdom. ISBN: 1584500778.
- Steven M. LaValle (2006). *Planning Algorithms.* Cambridge University Press.
  - http://planning.cs.uiuc.edu/
- Funge (2004). *Artificial Intelligence for Computer Games: An Introduction*. A K Peters. ISBN 1-56881-208-6.
- Bourg; Seemann (2004). *AI for Game Developers*. O'Reilly & Associates. ISBN 0-596-00555-5.
- Millington (2005). *Artificial Intelligence for Games*. Morgan Kaufmann. ISBN 0-12-497782-0.

Motion planning

The main question:

Are two points connected with a path?



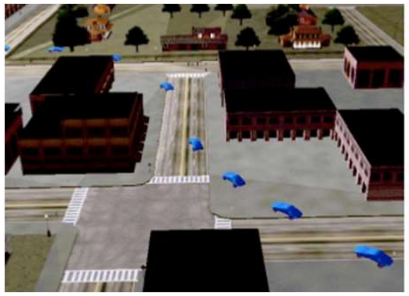
Objectives

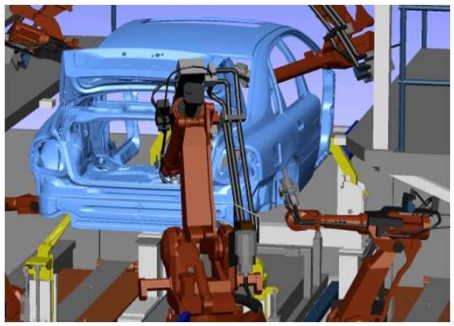
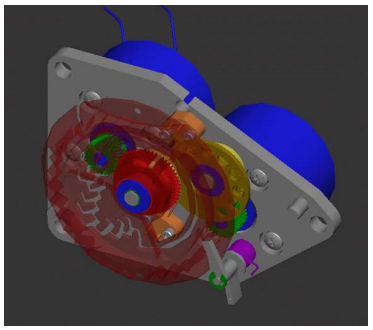
- Computation of movement strategies
  - Geometric paths
  - Trajectories

- Fulfill high level goals
  - Reach a target without collision with obstacles
  - Product assembly
  - …

Applications

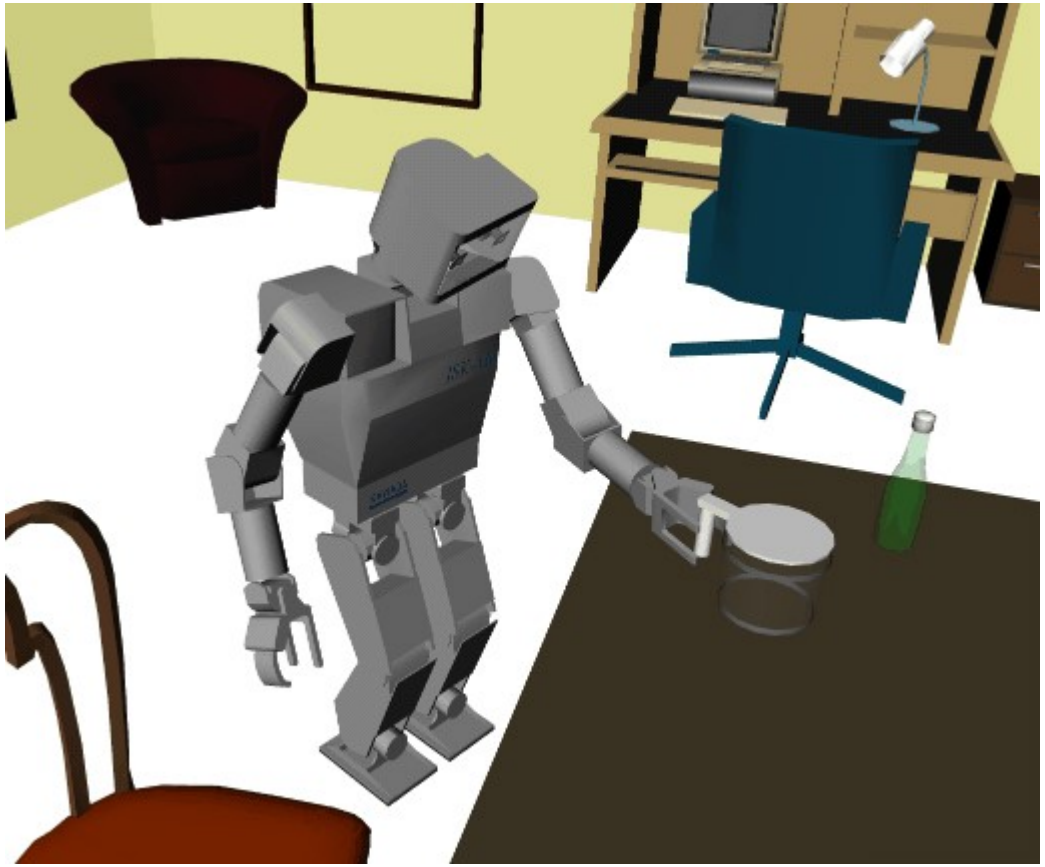
- Robot programming (painting, assembly)
- Autonomous vehicles (transport, planet exploration, military applications)
- Computer animation (video games, films)
- Surgery (implants)
- Molecular biology (medicine)



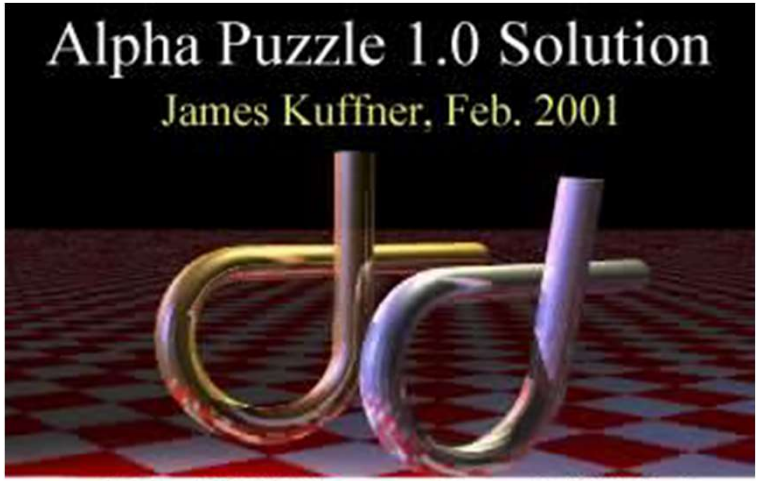



Examples

Examples



Examples



model by DSMFT group, Texas A&M Univ.
original model by Boris Yamrom, GE

The piano mover's problem

- Given a piano and an environment populated with obstacles, can we move the piano from one position / orientation (called configuration) to another without colliding obstacles?

Définition

- Given:
  - A mechanical system:
    - » Geometry
    - » Mobility
  - An environment
  - Two configurations: initial (source) and final (target)

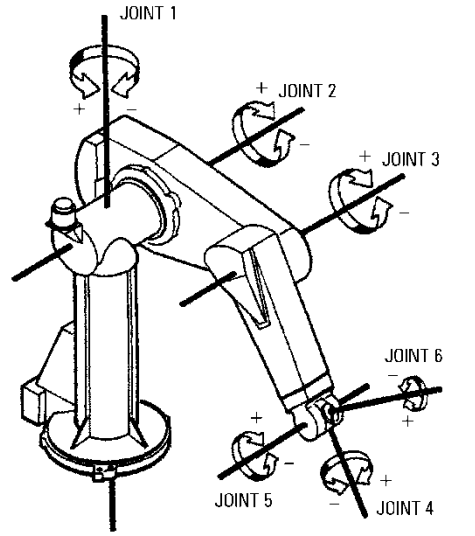
- We want to compute a path, without collision, joining the initial and final configurations.

Definition of a configuration

- A configuration gives the positions of all points of a mechanical system in a given coordinate system.
- A configuration is a vector of generalized coordinates.

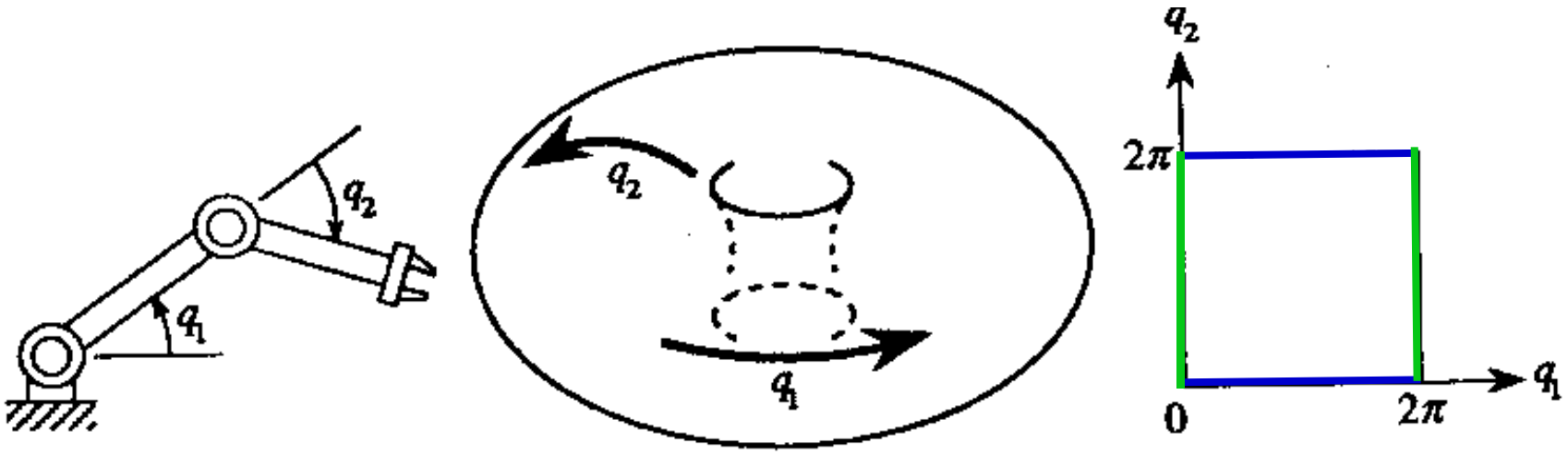
$$\begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \\ q_6 \end{bmatrix}$$

Figure 2 The unimate PUMA 562 robot arm



The configuration space (C-space)

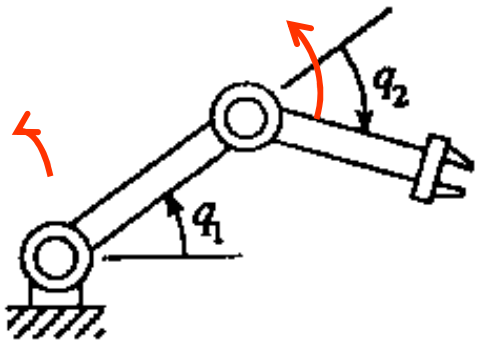
- Contains all possible configurations of a given mechanical system
- Formalism described by [Lozano-Perez '79] (C-space)

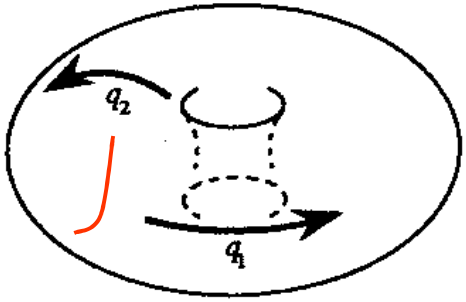


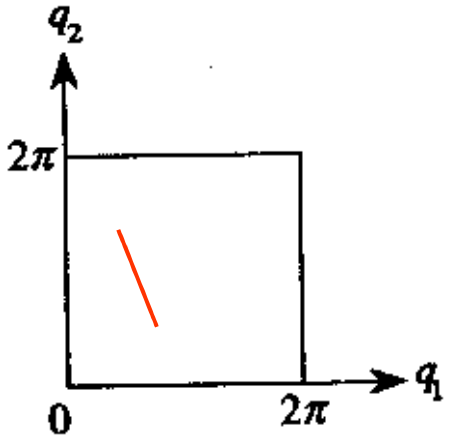
Path vs trajectory

- Path: a continuous sequence of configurations
- Trajectory: temporal parameterization of a path

$$q(t) = \begin{bmatrix} q_1(t) \\ q_2(t) \\ \dots \\ q_n(t) \end{bmatrix}, \; t \in [0, T]$$



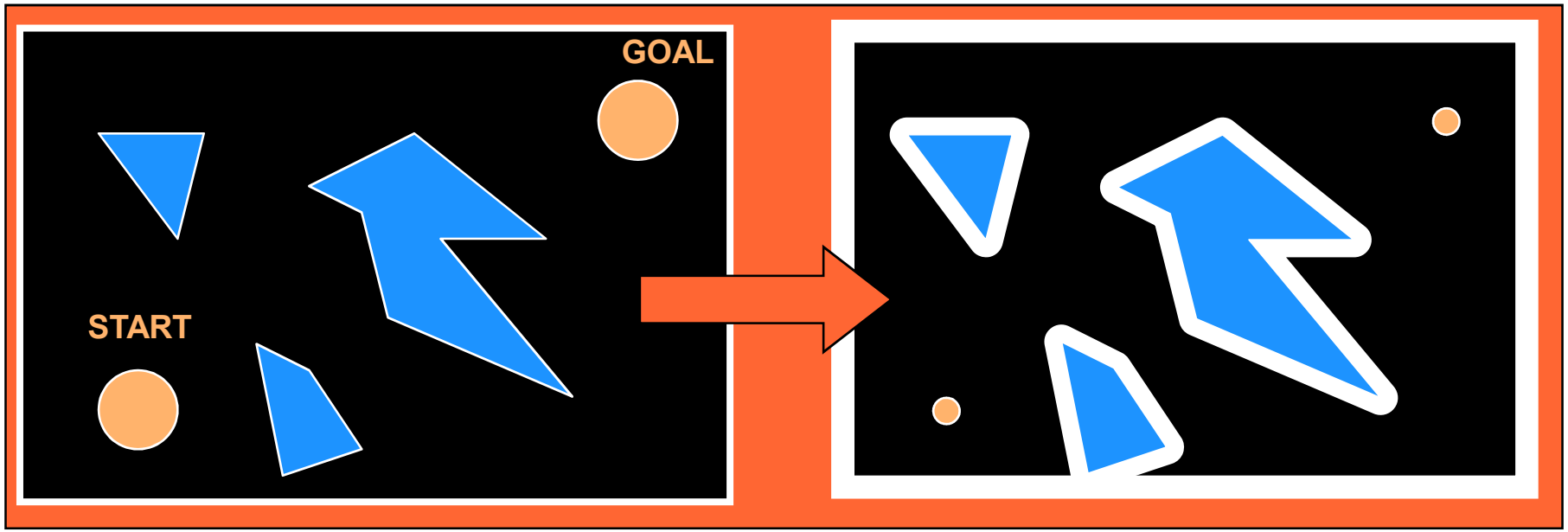




The free space (C-free)

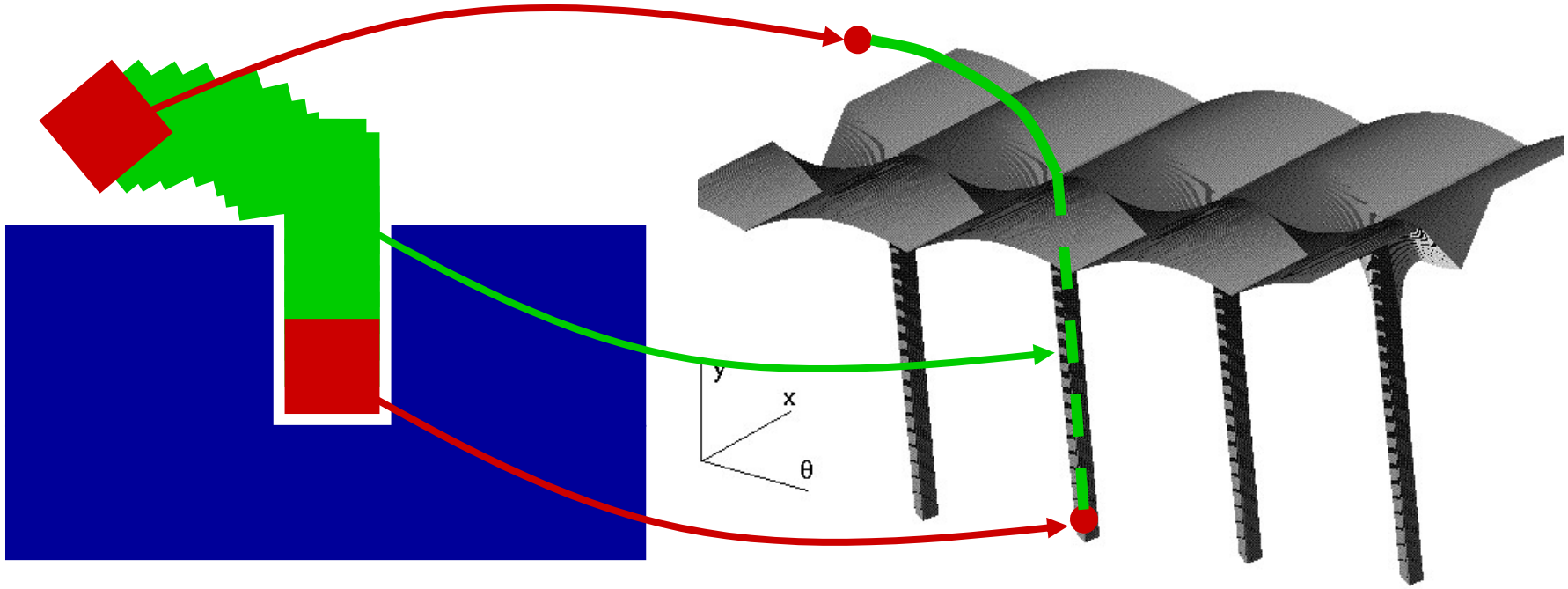
Example: 2D navigation

$$q = \begin{bmatrix} x \\ y \end{bmatrix}$$



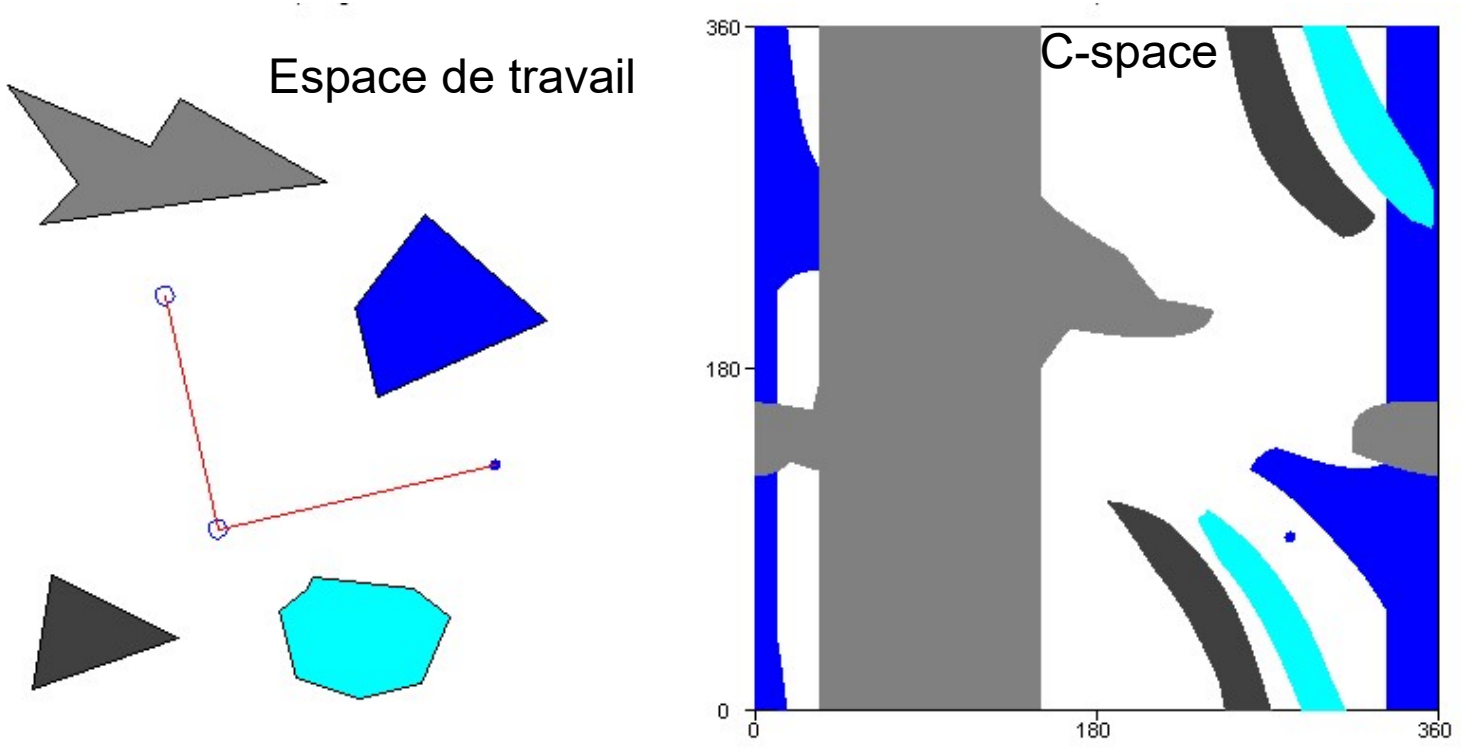
C-space representation

$$C_{space} = C_{free} + C_{obst}$$



C-space representation

$$C_{space} = C_{free} + C_{obst}$$



Images created by C-space Java applet : http://ford.ieor.berkeley.edu/cspace/

Main approaches

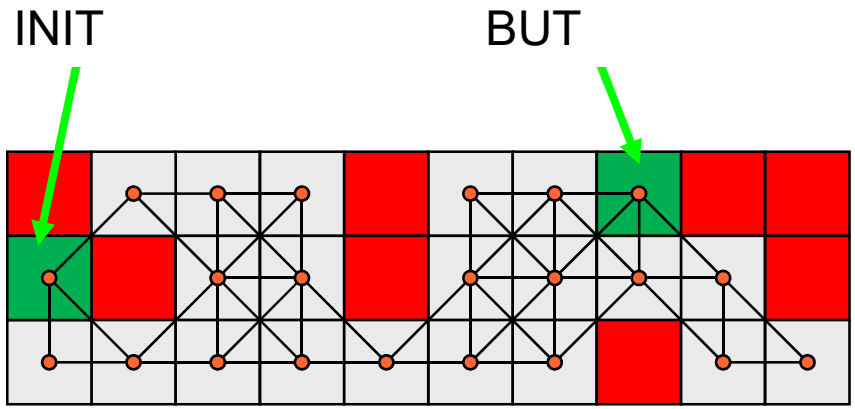
- Deterministic search
  - Approximate: grids
  - Exact: Cell decompositions
- Navigation functions: potential fields
- Probabilistic sampling: exploring high dimensional spaces
  - PRM (Probabilistic RoadMaps)
  - RRT (Rapidly exploring Random Trees)

Main approaches

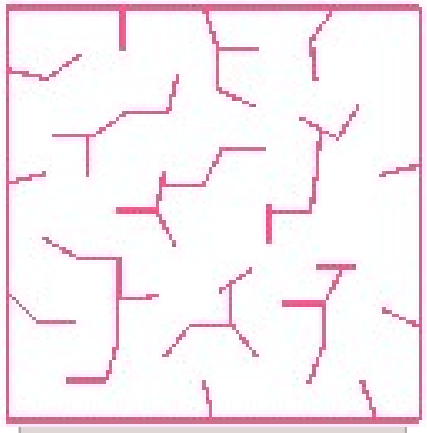
- **Deterministic search**
  - **Approximate: grids**
  - **Exact: Cell decompositions**
- Navigation functions: potential fields
- Probabilistic sampling: exploring high dimensional spaces
  - PRM (Probabilistic RoadMaps)
  - RRT (Rapidly exploring Random Trees)

Approximate methods: Grids

- Approximate representation of the C-space (C-free and C-obstacle)
- Each cell is marked free or obstacle

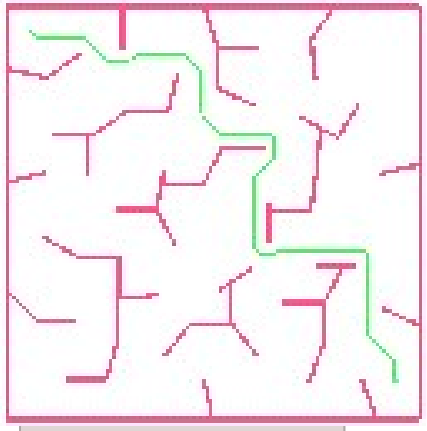


PLANNING: (Dijkstra, A*…)



Projected map

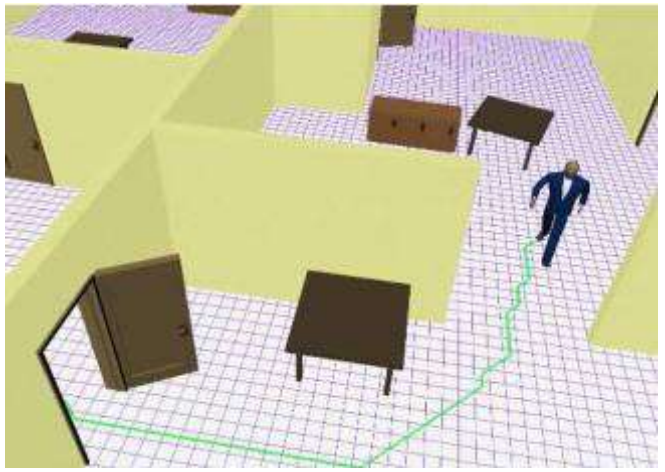
Extended obstacles

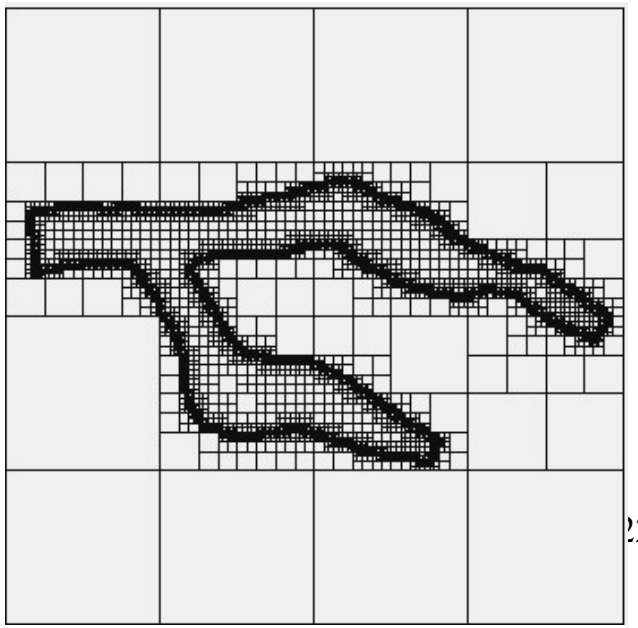


Planned path

Grids

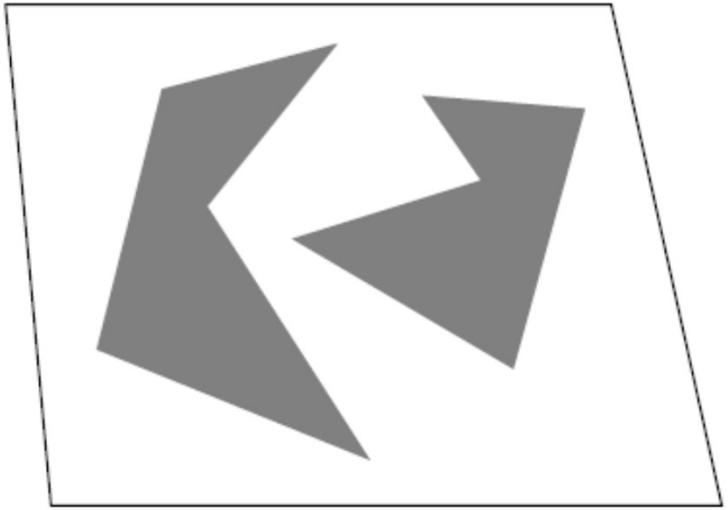
- Pros:
  - Easy to implement
  - Fast to compute
- Cons:
  - Lack of precision
  - Huge search time in high dimensional problems
- Can be improved
  - Quadtree / Octree
  - More generally: hierarchical representations



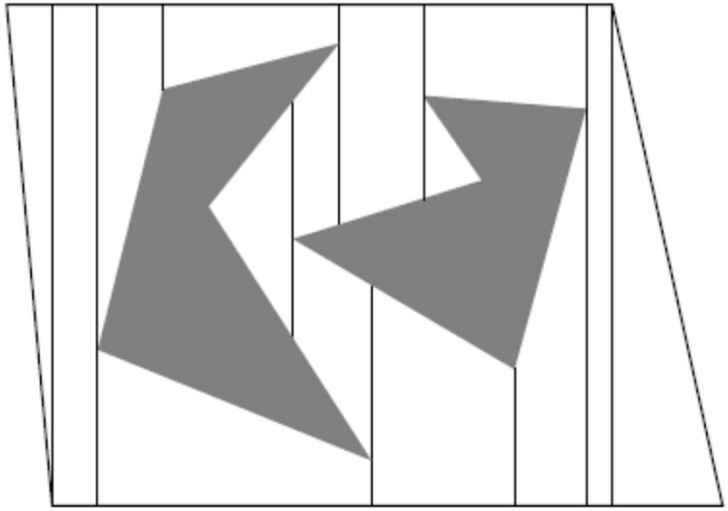


Exact methods: cell decomposition

- Principle:
  - Decompose C-free into cells of simple geometric shapes (triangles, quads…)

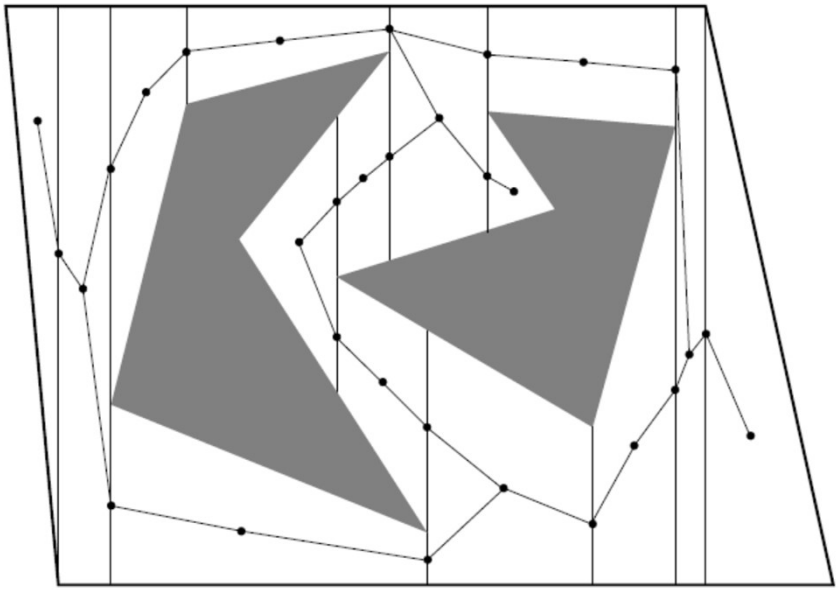


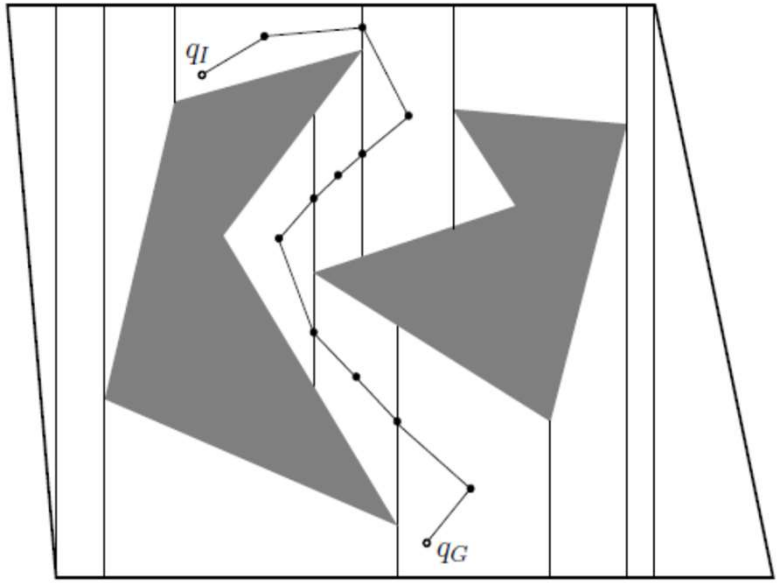
- Example:
  - Vertical decomposition



Exact cell decomposition

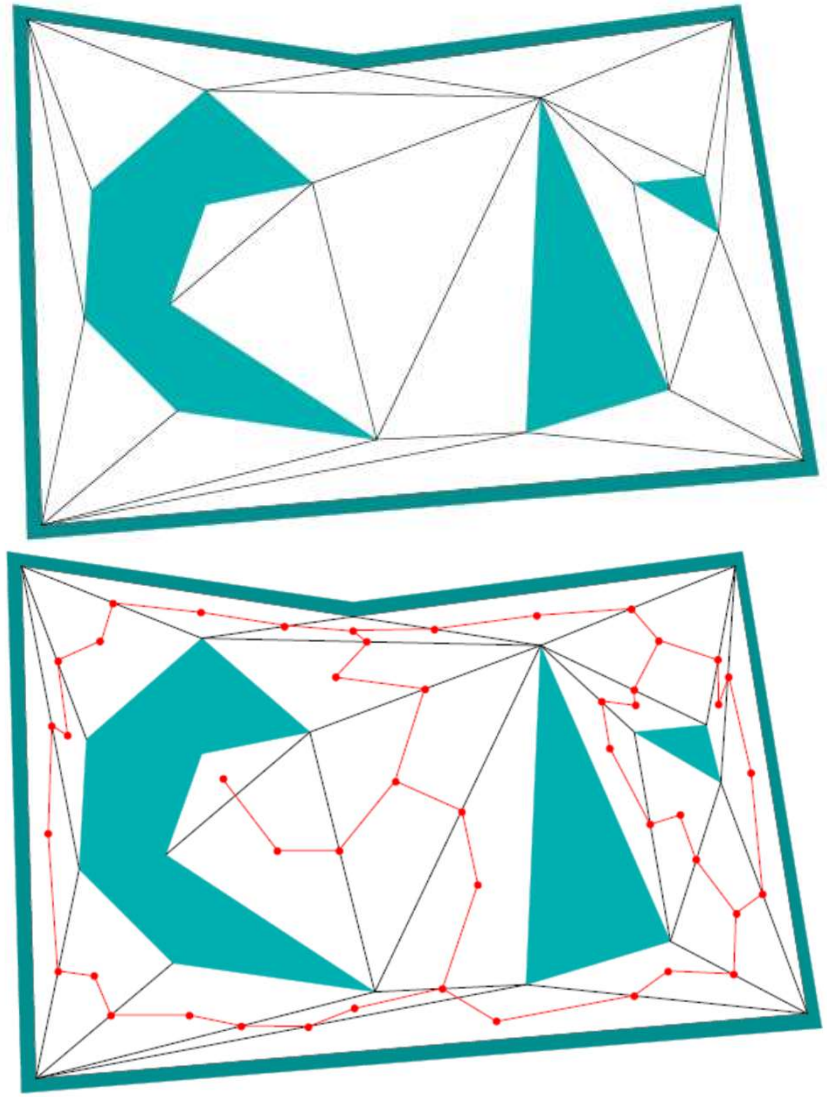
- A roadmap, capturing C-free topology, is deduced from the decomposition
- Initial and final configurations are connected to the graph and a path can be planned





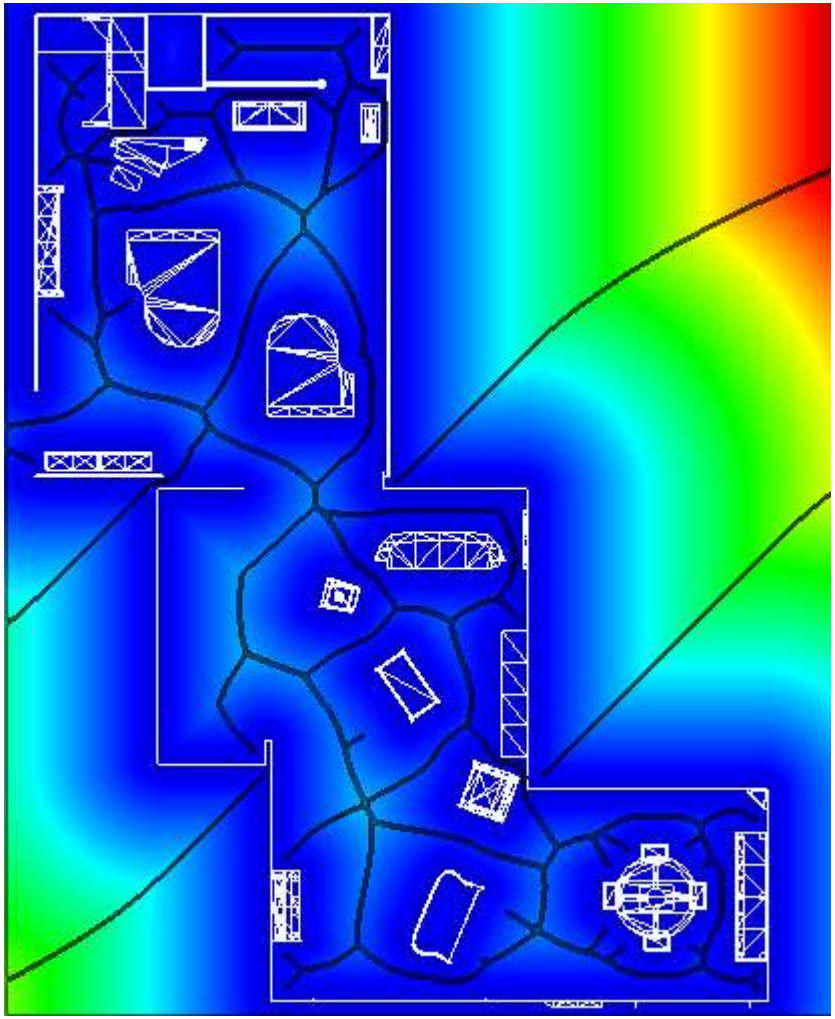
Exact cell decomposition

- Constrained Delaunay triangulation
  - Computes a partition of the plane with triangles
  - Obstacles borders are constrained segments
- Some improvements
  - Computation of bottlenecks



Exact methods: Voronoi diagrams

- Generalized Voronoi diagrams
- Useful in robotics
  - Computation of safest path (far from obstacles)

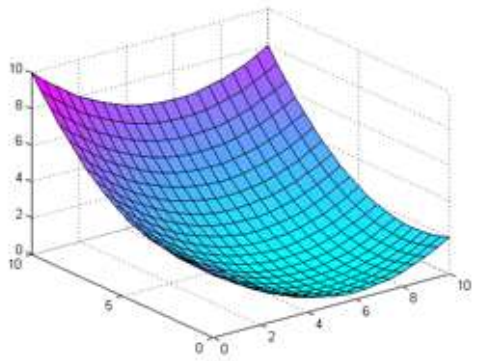


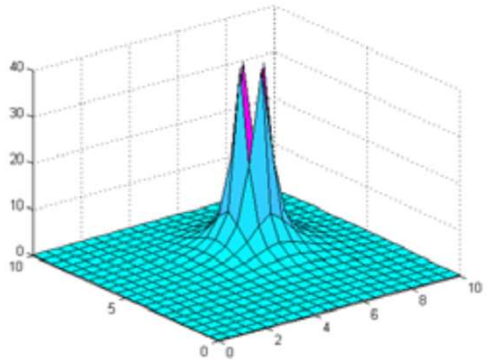
Main approaches

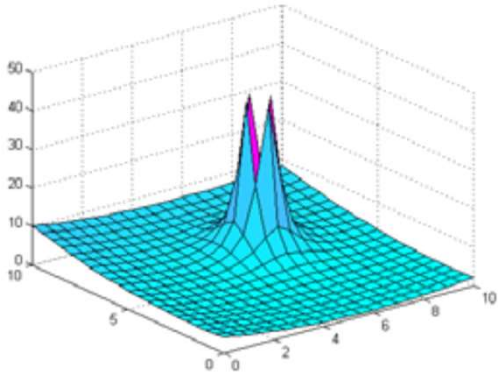
- Deterministic search
  - Approximate: grids
  - Exact: Cell decompositions
- **Navigation functions: potential fields**
- Probabilistic sampling: exploring high dimensional spaces
  - PRM (Probabilistic RoadMaps)
  - RRT (Rapidly exploring Random Trees)

Potential fields

- Function defined on C-free
- Ideally:
  - Push mobile far from obstacles
  - Attract mobile toward the goal





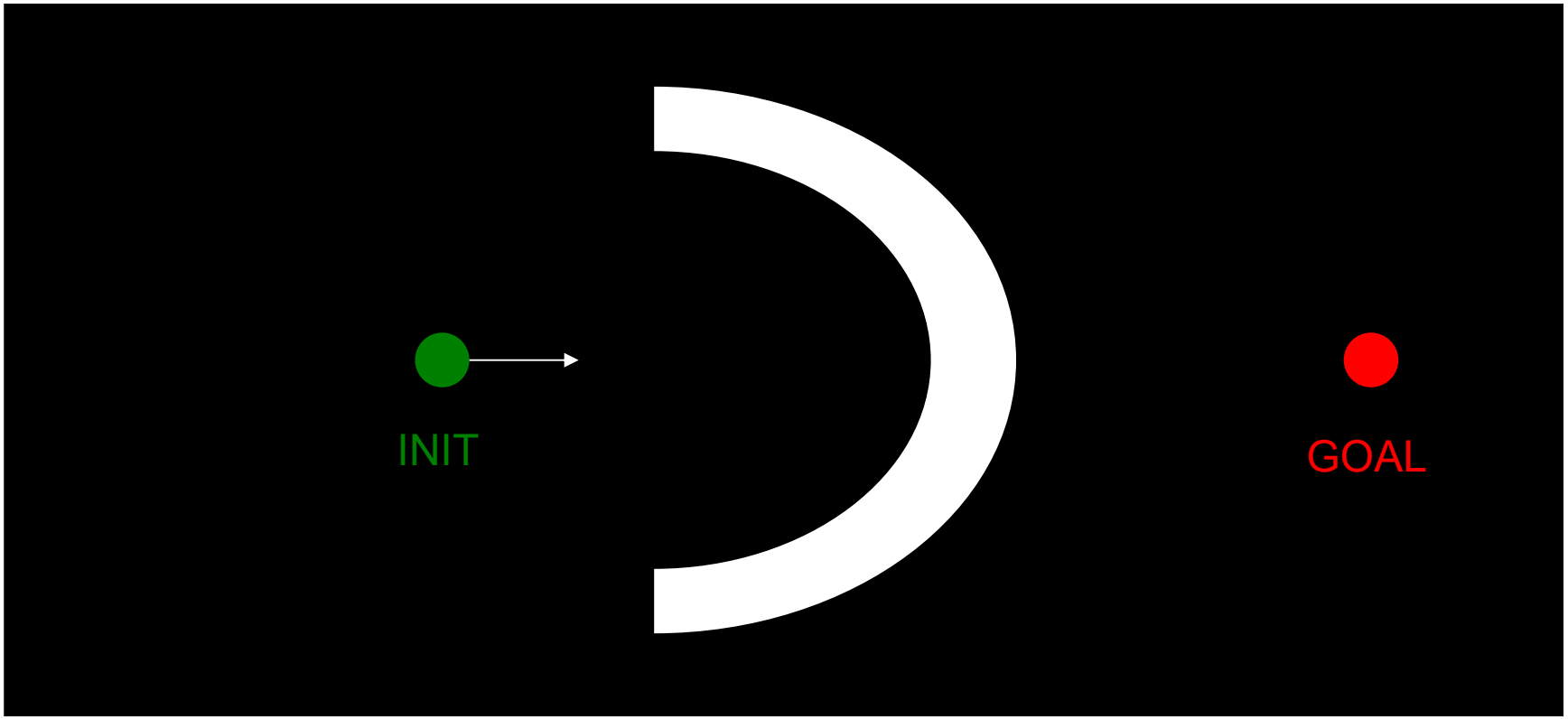


Potential fields

- An attractive solution for planning problems
  - Paths are not planned but generated in real time
  - Planning and control are merged in a function
  - Paths are smoothed
  - Potential fields can be updated dynamically
  - To reduce computation, only local obstacles can be taken into account

Potential fields

- **But: local minima problem**



Potential fields

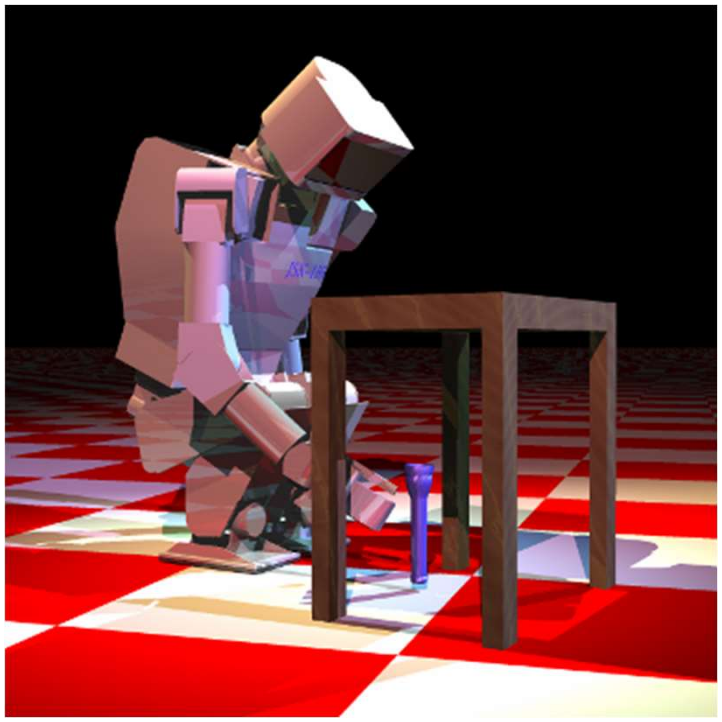
- Solutions to the local minima problem:
  - Backward chaining
  - Random walk
  - Use a roadmap
- Pre-computation:
  - Potential field definition
  - Using a regular grid to store potentials
- Planning:
  - Search in the grid by using potential fields as a heuristic

Main approaches

- Deterministic search
  - Approximate: grids
  - Exact: Cell decompositions
- Navigation functions: potential fields
- **Probabilistic sampling: exploring high dimensional spaces**
  - **PRM (Probabilistic RoadMaps)**
  - **RRT (Rapidly exploring Random Trees)**

Planning in high dimensional space

- Ideal: a complete planner
  - Guaranties to find a solution in a finite time if it exists
  - Indicates that no solution exists in a finite time
  - Remark: completeness implies the computation of at least one path among all possible paths
- Problem:
  - PSPACE complete
- Remark:
  - Exact deterministic methods are complete
  - Approximate deterministic methods are complete in representation



Planning in high dimensional space

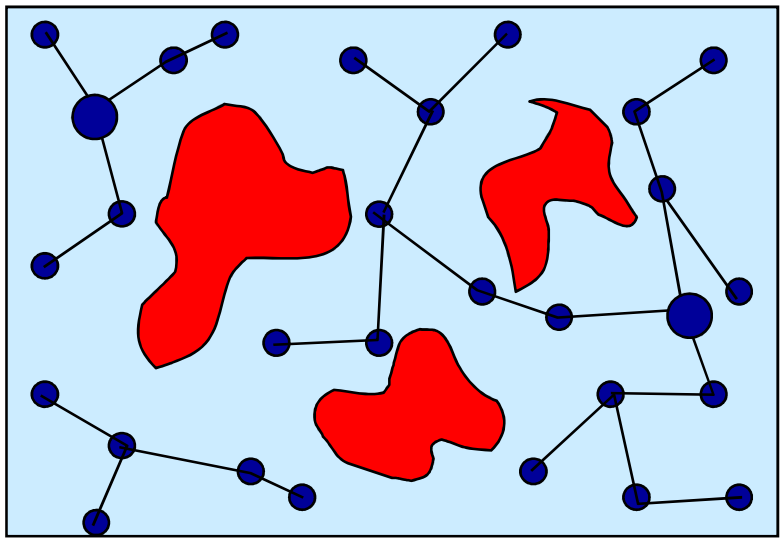
- Solutions
  - Decrease the number of dimensions
  - Limit the number of possible solutions
    - Add constraints
    - Reduce search volume
  - Sacrifice optimality
  - Sacrifice completeness

Planning in high dimensional space

- Key idea: random exploration of the free space to capture the result
- Easy way of capturing C-free topology and faster than an exhaustive exploration
- The problem?
  - Sacrifice of completeness and optimality
  - *Compromise between quality and performances*

PRM : Probabilistic RoadMaps

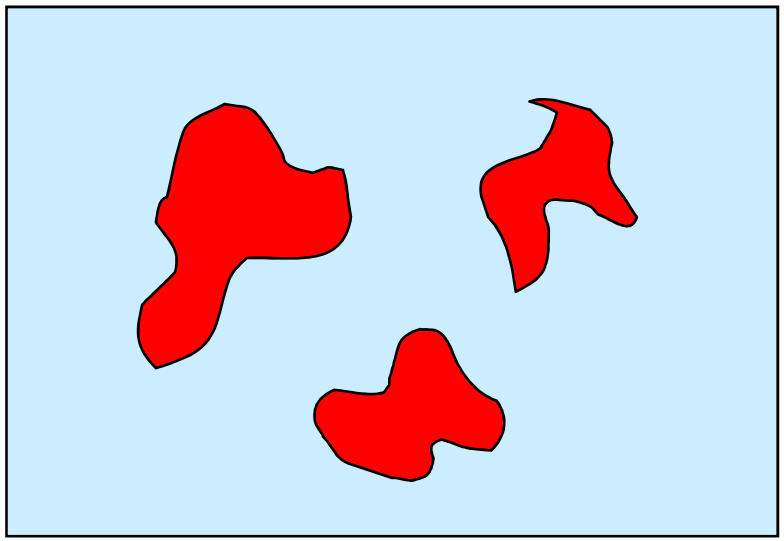
- Three components
  - Collision detection
  - Local method
  - Sampling
- Two steps
  - Exploration phase
  - Query phase
- Key idea: random exploration of C-free to capture its topology in a roadmap
- Complete in infinite time: probabilistic completeness



The exploration phase
Computation of the roadmap

- Iterative algorithm

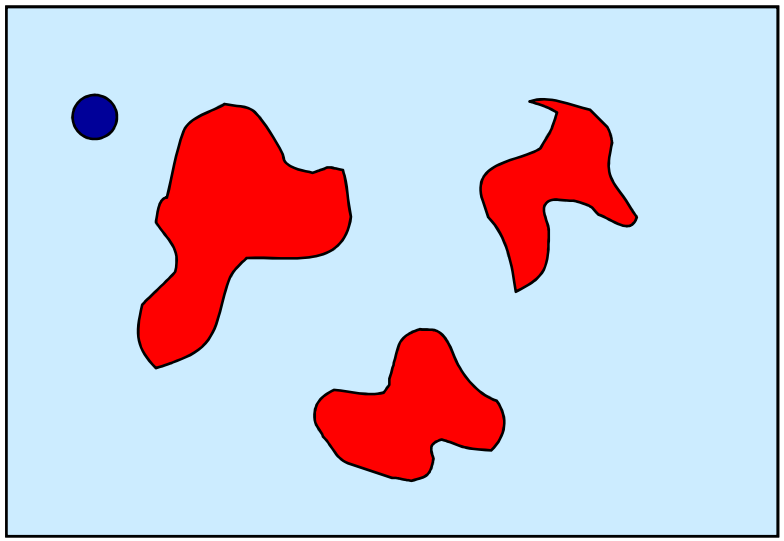
1. Computation of random configurations
   - ➢ Collision detection
2. Liking configuration
   - ➢ Local method
     - » Interpolation between configurations
     - » Collision detection
3. Return to 1



The exploration phase
Computation of the roadmap

- Iterative algorithm

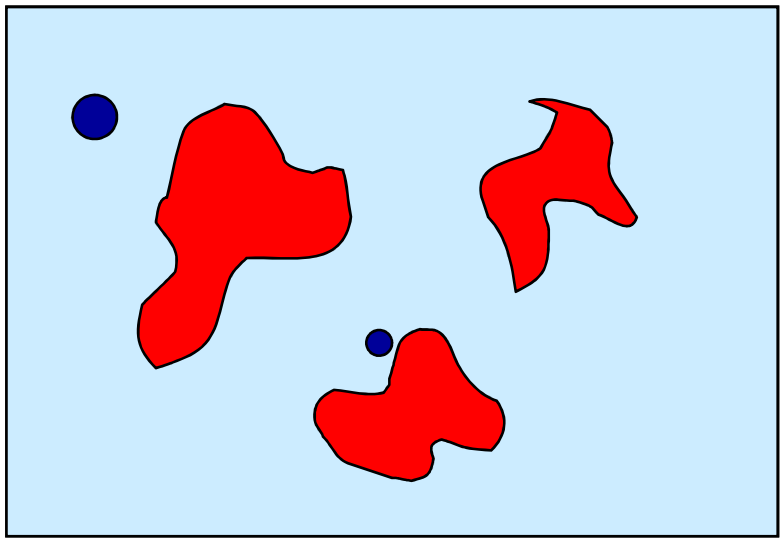
1. Computation of random configurations
   - ➢ Collision detection
2. Liking configuration
   - ➢ Local method
     - » Interpolation between configurations
     - » Collision detection
3. Return to 1



The exploration phase
Computation of the roadmap

- Iterative algorithm

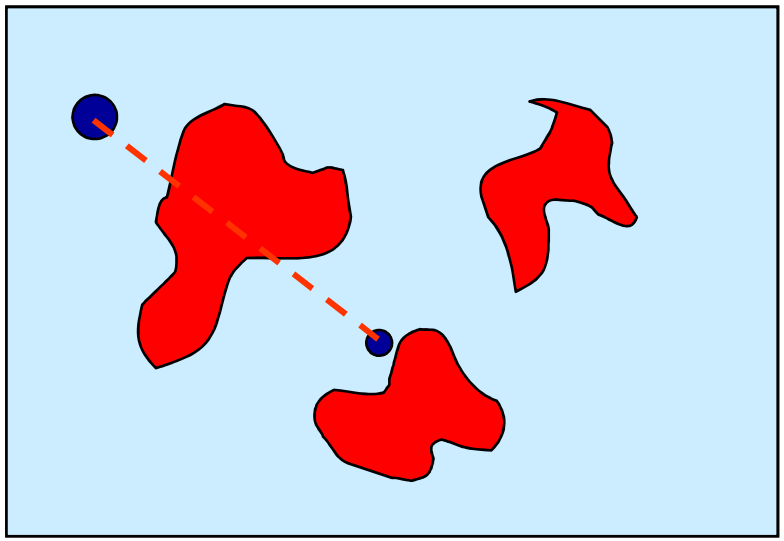
1. Computation of random configurations
   - ➢ Collision detection
2. Liking configuration
   - ➢ Local method
     - » Interpolation between configurations
     - » Collision detection
3. Return to 1



The exploration phase
Computation of the roadmap

- Iterative algorithm

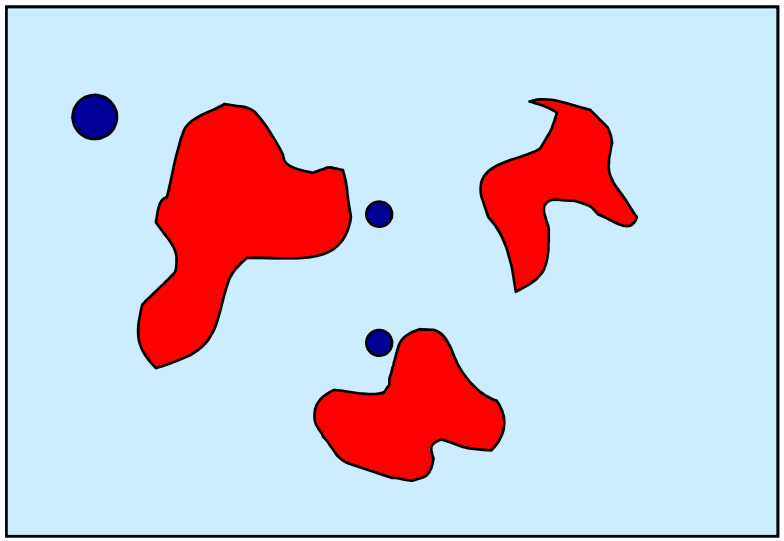
1. Computation of random configurations
   - ➢ Collision detection
2. Liking configuration
   - ➢ Local method
     - » Interpolation between configurations
     - » Collision detection
3. Return to 1



The exploration phase
Computation of the roadmap

- Iterative algorithm

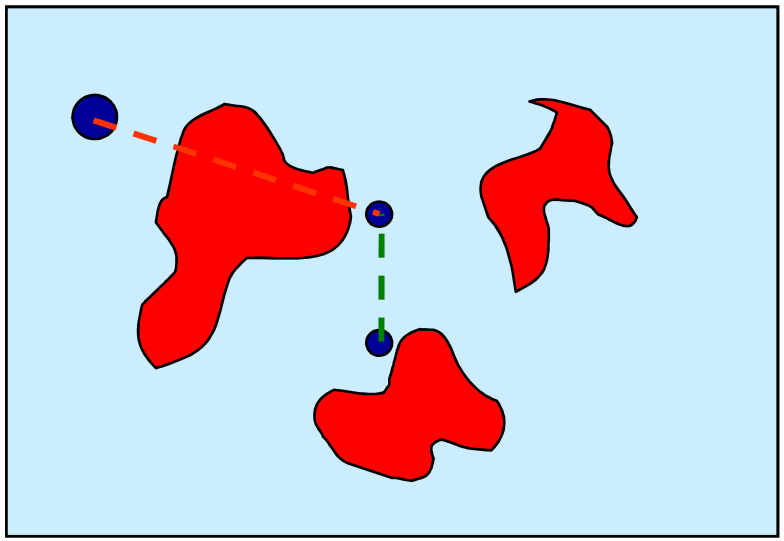
1. Computation of random configurations
   - ➢ Collision detection
2. Liking configuration
   - ➢ Local method
     - » Interpolation between configurations
     - » Collision detection
3. Return to 1



The exploration phase
Computation of the roadmap

- Iterative algorithm

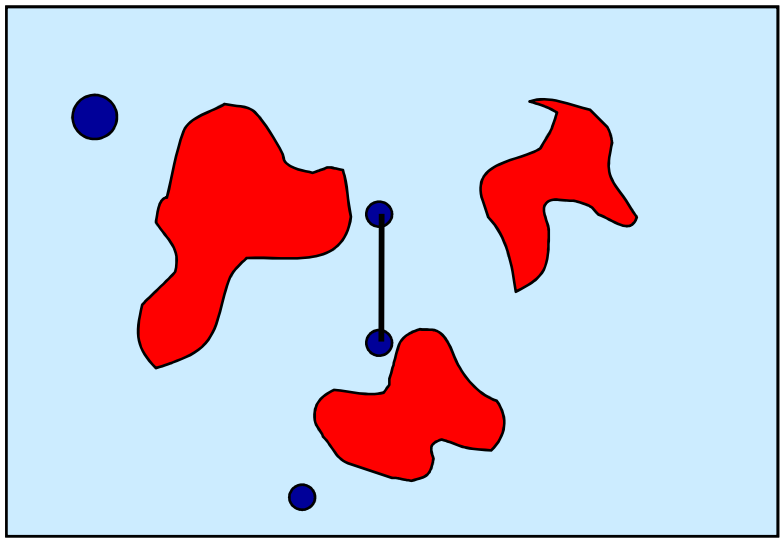
1. Computation of random configurations
   - ➢ Collision detection
2. Liking configuration
   - ➢ Local method
     - » Interpolation between configurations
     - » Collision detection
3. Return to 1



The exploration phase
Computation of the roadmap

- Iterative algorithm

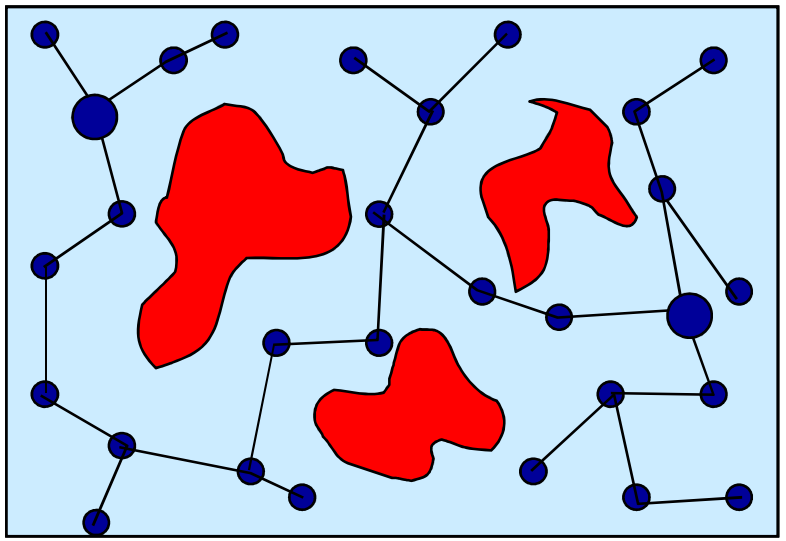
1. Computation of random configurations
   - ➢ Collision detection
2. Liking configuration
   - ➢ Local method
     - » Interpolation between configurations
     - » Collision detection
3. Return to 1



The exploration phase
Computation of the roadmap

- Iterative algorithm

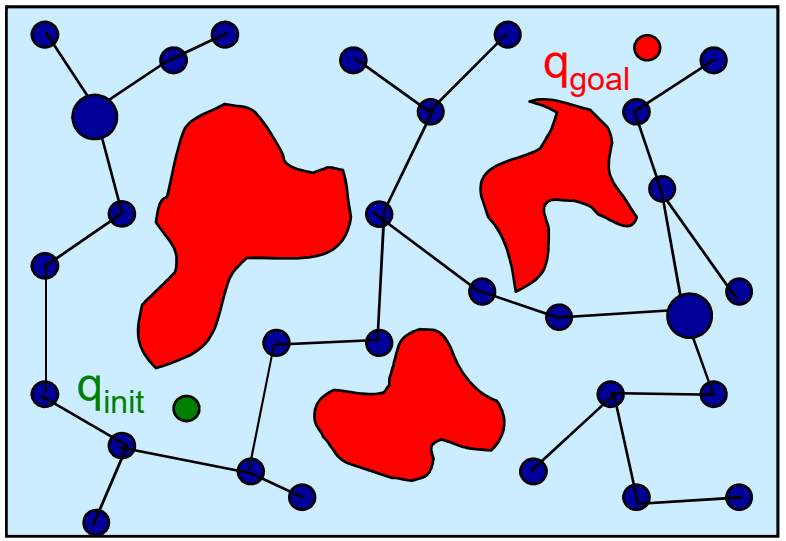
1. Computation of random configurations
   - ➢ Collision detection
2. Liking configuration
   - ➢ Local method
     - » Interpolation between configurations
     - » Collision detection
3. Return to 1



The query phase

- The PRM is used to solve queries

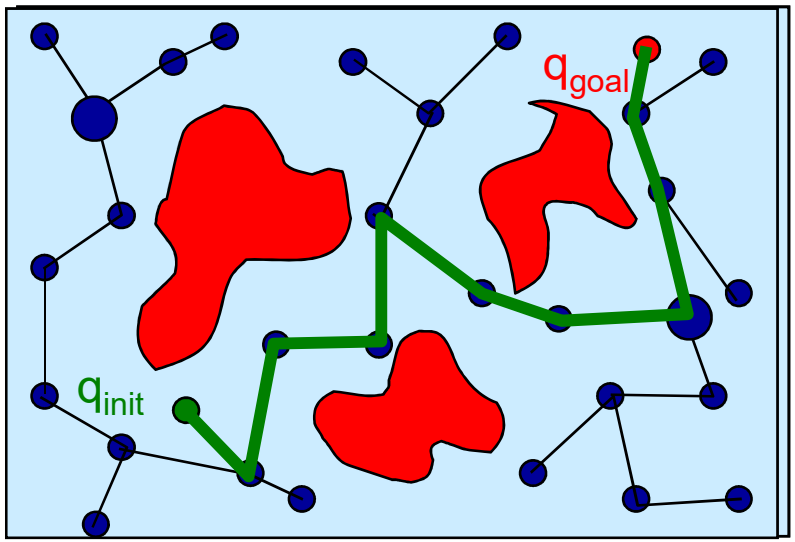
1. Connect initial and final configurations to the roadmap
2. If nodes belong to the same connected component, a solution exists
3. Search the solution in the graph
4. Optimize / smooth the path



The query phase

- The PRM is used to solve queries

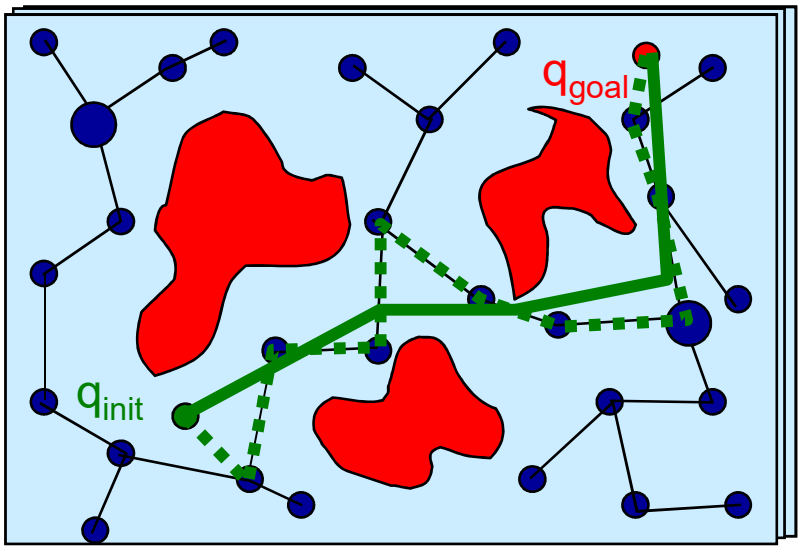
1. Connect initial and final configurations to the roadmap
2. If nodes belong to the same connected component, a solution exists
3. Search the solution in the graph
4. Optimize / smooth the path



The query phase

- The PRM is used to solve queries

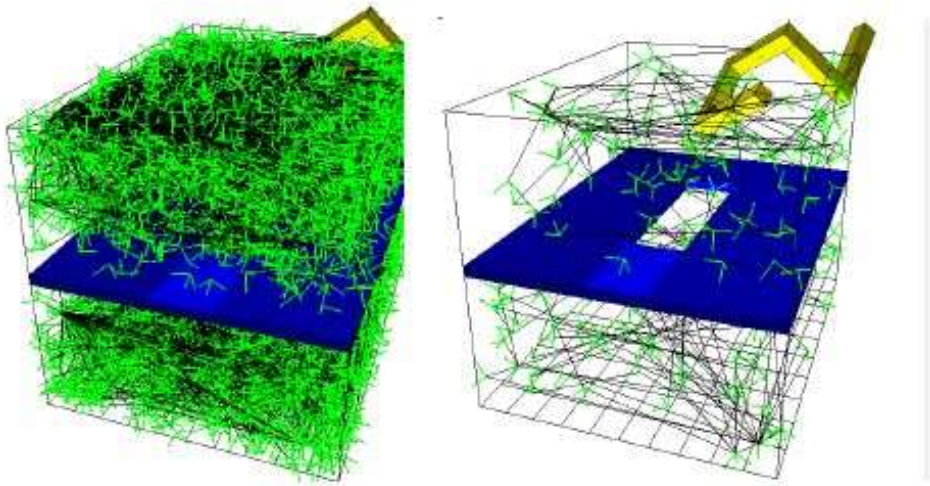
1. Connect initial and final configurations to the roadmap
2. If nodes belong to the same connected component, a solution exists
3. Search the solution in the graph
4. Optimize / smooth the path



Example

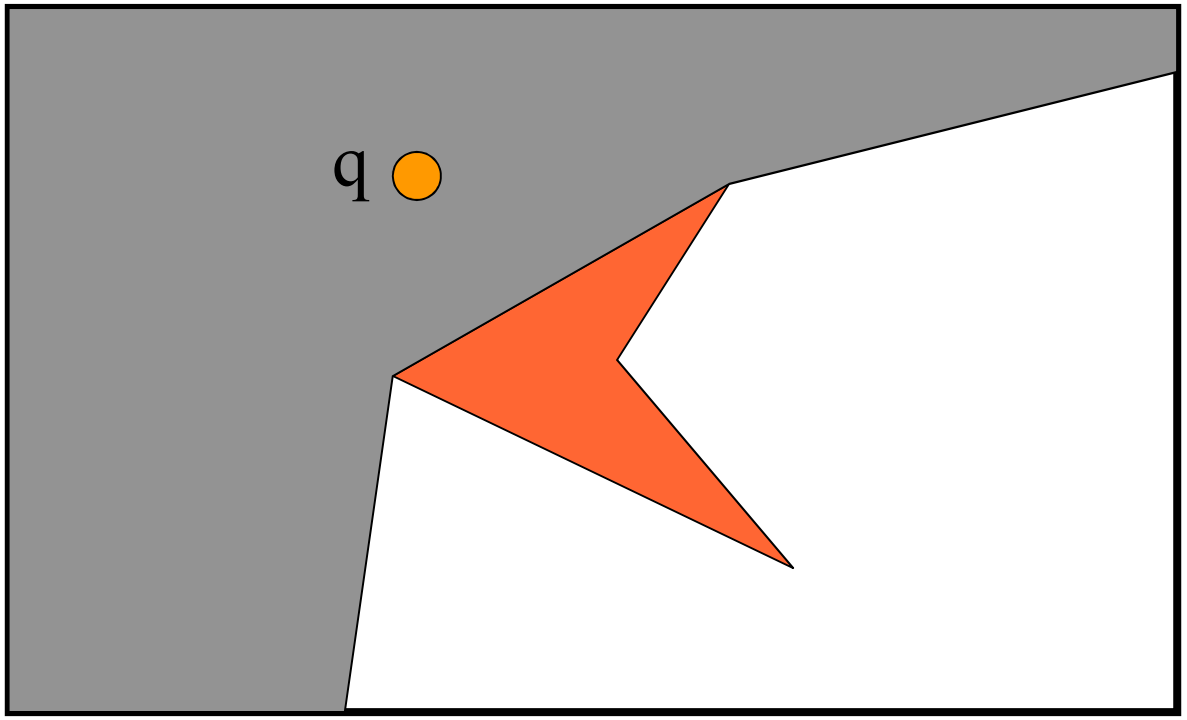
Sampling strategies

- Planning in very constrained environements generates large roadmaps
  - Construction is time consuming
  - Search is slow
- Use of sampling strategies
  - Visibility PRMs
  - Gaussian PRMs



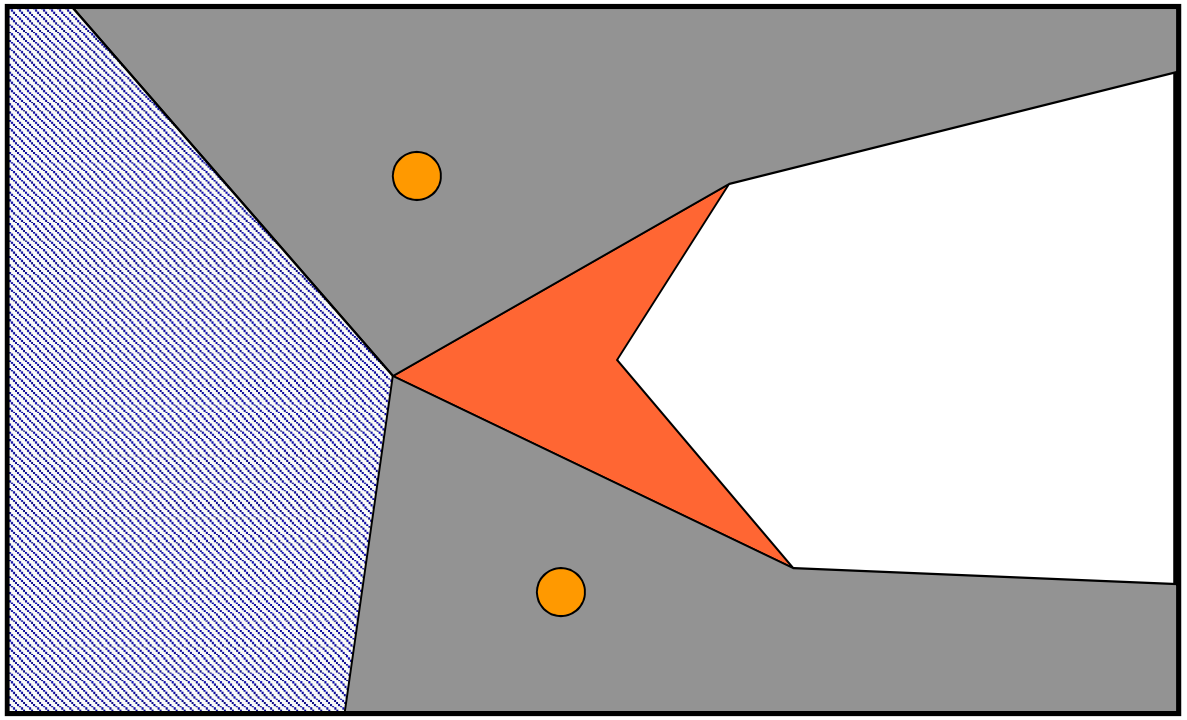
Visibility-PRM

Visibility domain of a configuration q



Visibility-PRM

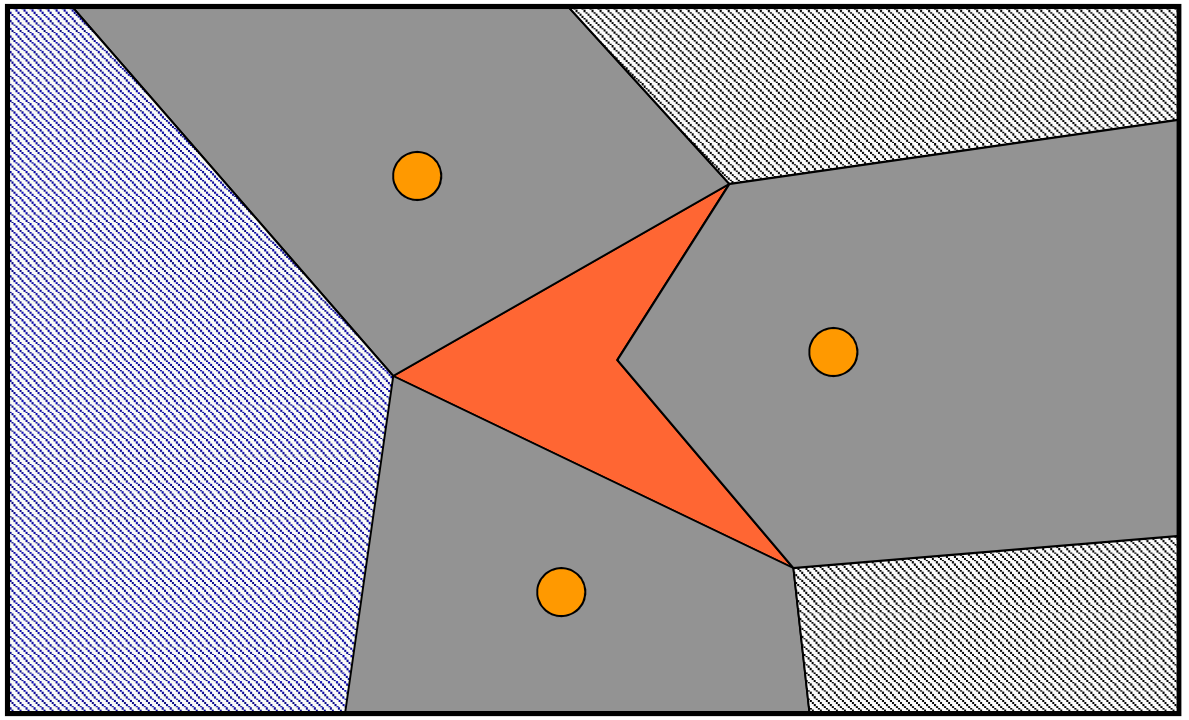
A new configuration is kept if it is outside the visiblity domain of other configurations



Visibility-PRM

A new configuration is kept if it is outside the visiblity domain of other configurations

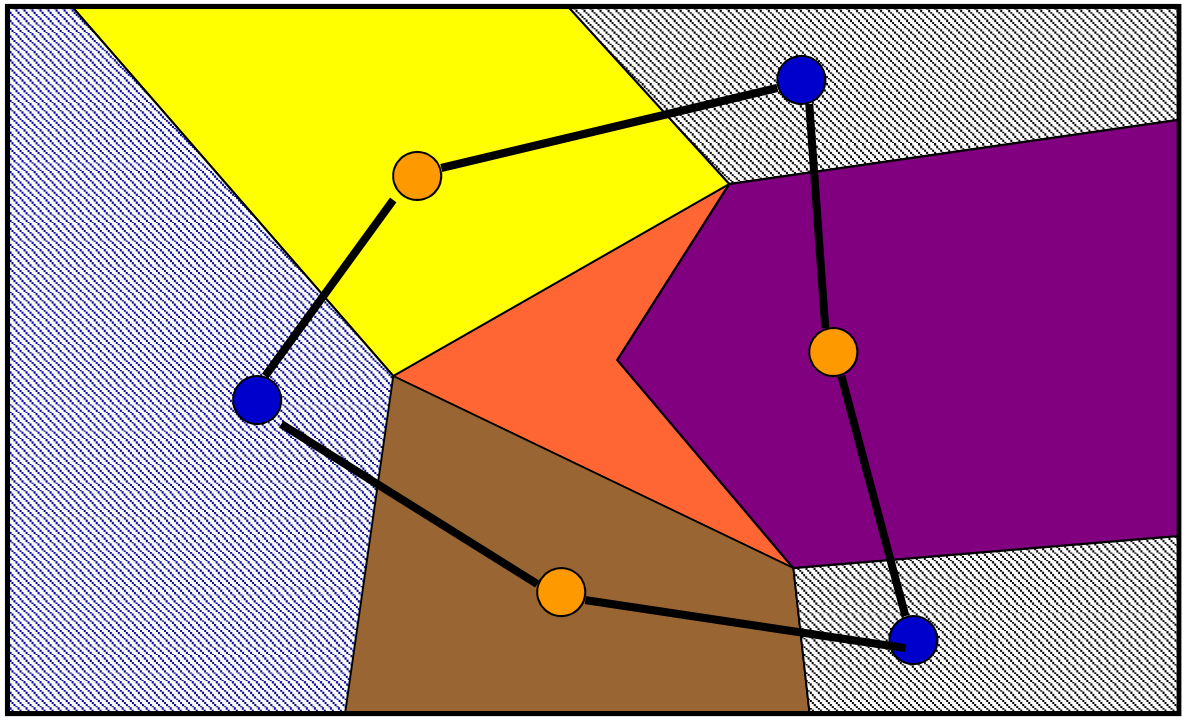
Those configurations are “guardians”



Visibility-PRM

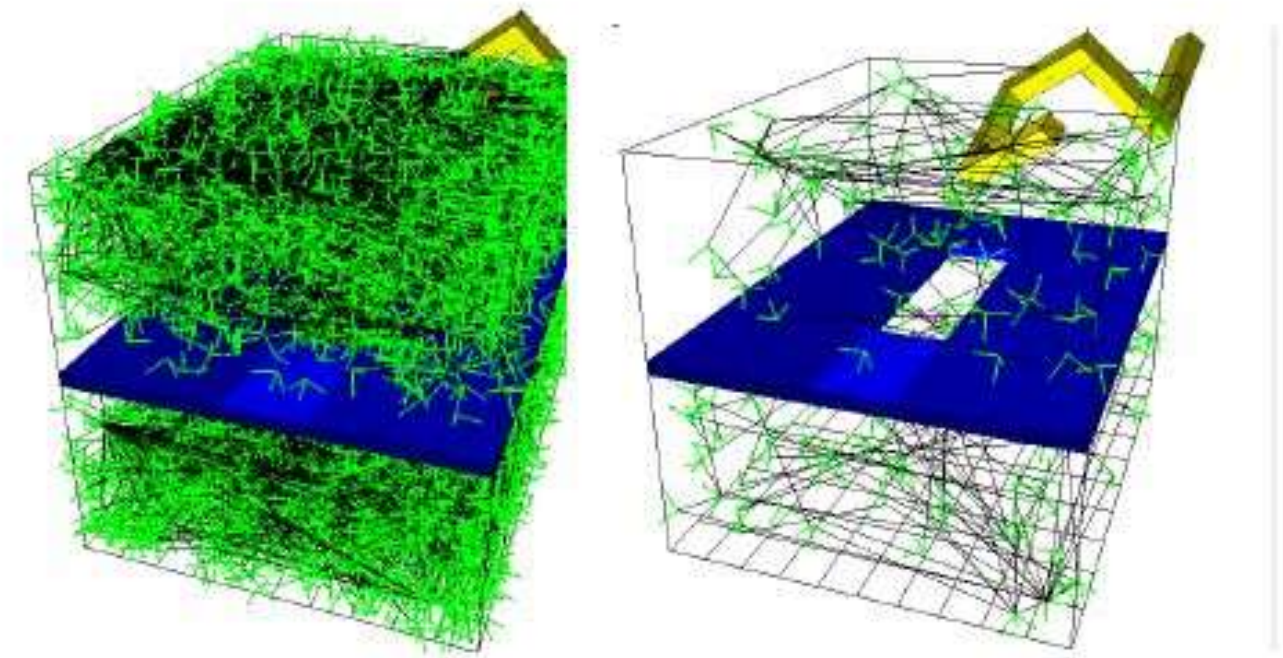
A new configuration is kept if it is outside the visiblity domain of other configurations **or if it connects at least two gardians**

Those configurations are "connectors"



Visibility-PRM

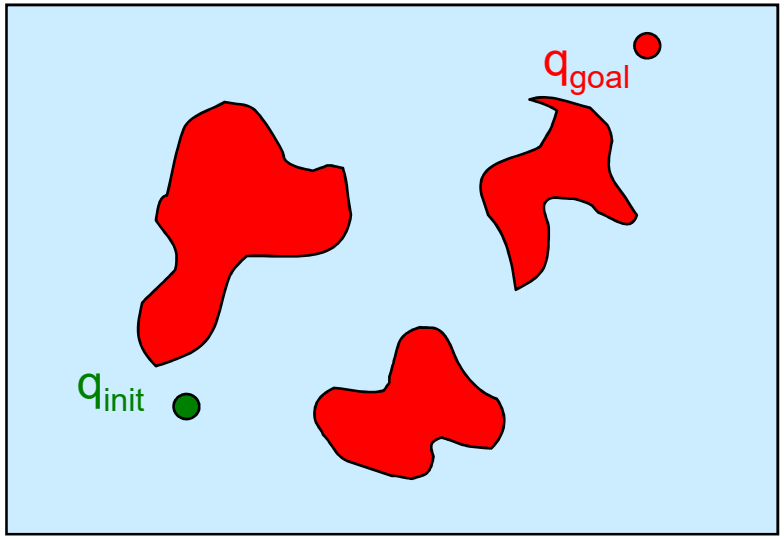
- Example
  - 6 dimensions configuration space



RRT:
Rapidly-exploring Random trees

- PRM is a “multiple queries” method: the same roadmap is used to solve several queries

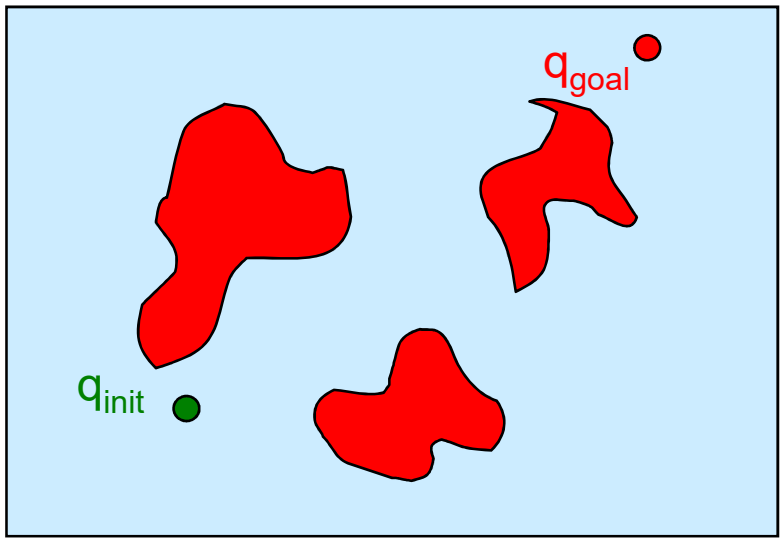
- RRT is a “single query” method: a datastructure is computed to solve one query without preliminary exploration of the C-space



RRT: Rapidly-exploring Random trees

- An iterative algorithm

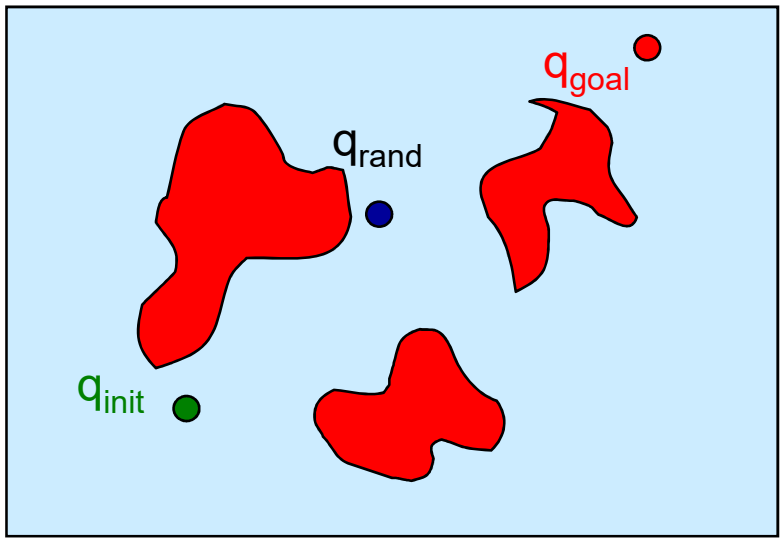
1. Compute q_{rand}
2. Connect to q_{near}
3. Insert q_{new}
4. Go back to 1



RRT:
Rapidly-exploring Random trees

- An iterative algorithm

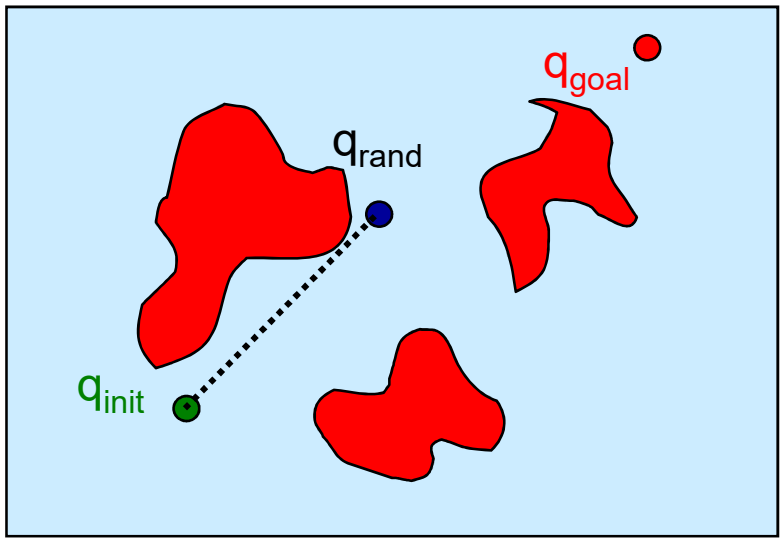
1. Compute q_{rand}
2. Connect to q_{near}
3. Insert q_{new}
4. Go back to 1



RRT:
Rapidly-exploring Random trees

- An iterative algorithm

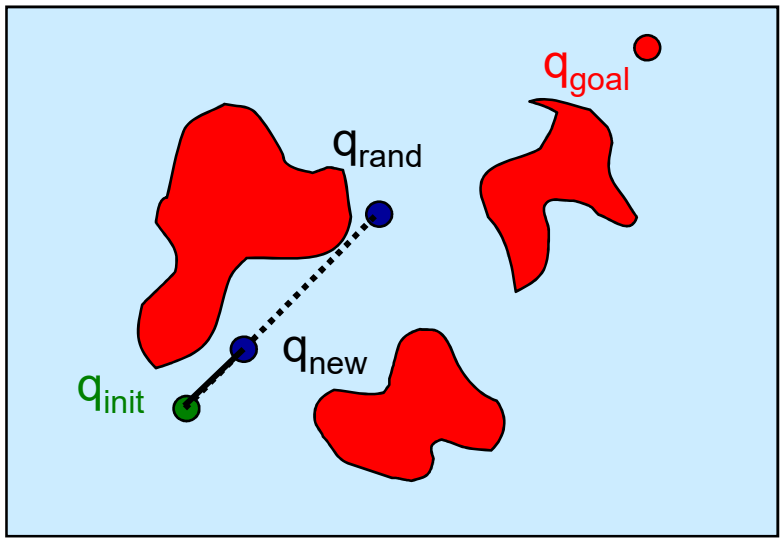
1. Compute q_{rand}
2. Connect to q_{near}
3. Insert q_{new}
4. Go back to 1



RRT:
Rapidly-exploring Random trees

- An iterative algorithm

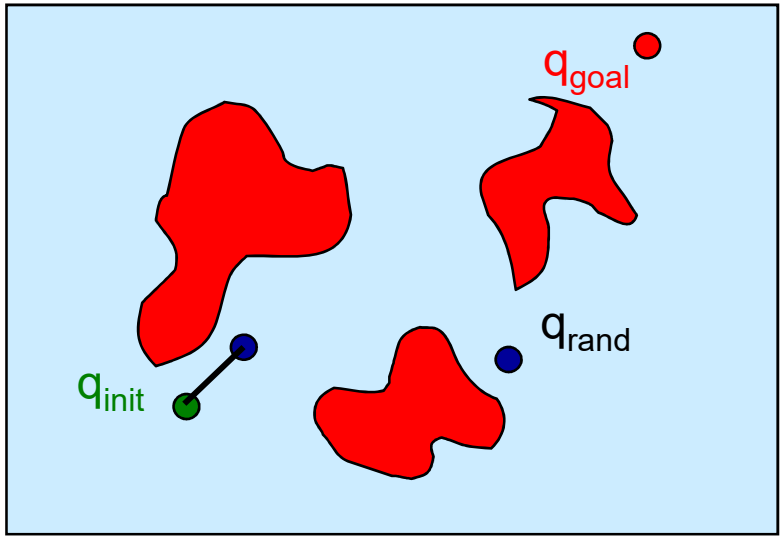
1. Compute q_{rand}
2. Connect to q_{near}
3. Insert q_{new}
4. Go back to 1



RRT:
Rapidly-exploring Random trees

- An iterative algorithm

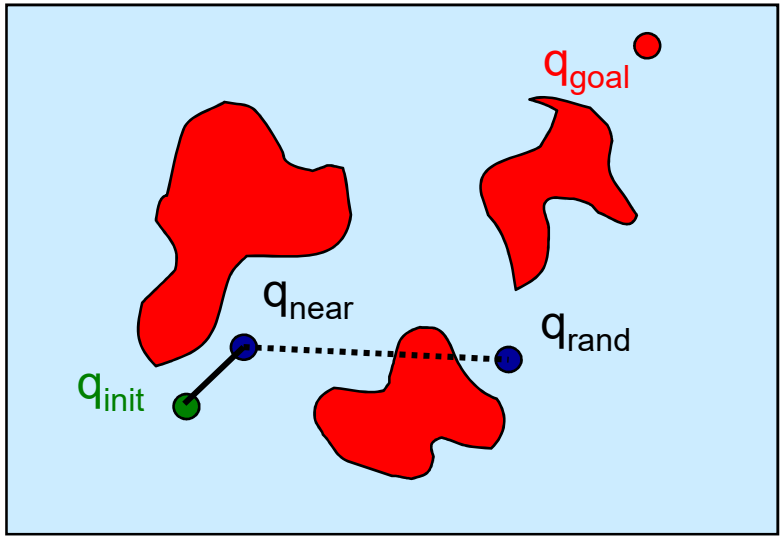
1. Compute q_{rand}
2. Connect to q_{near}
3. Insert q_{new}
4. Go back to 1



RRT:
Rapidly-exploring Random trees

- An iterative algorithm

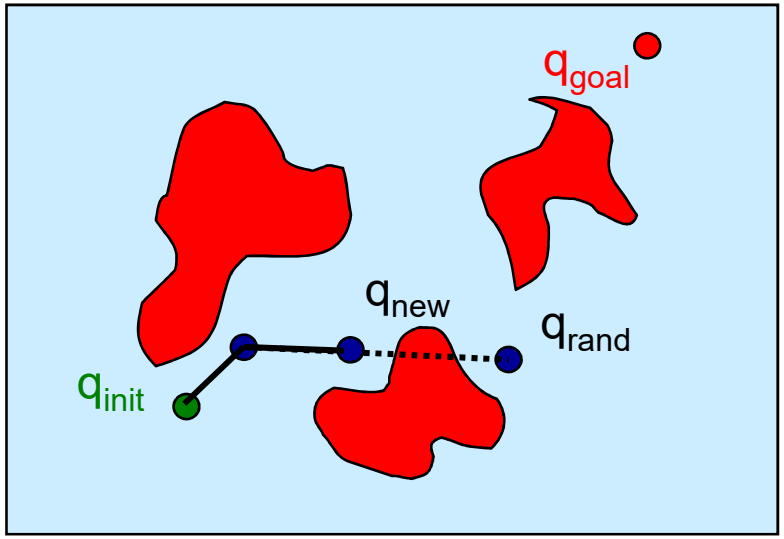
1. Compute q_{rand}
2. Connect to q_{near}
3. Insert q_{new}
4. Go back to 1



RRT:
Rapidly-exploring Random trees

- An iterative algorithm

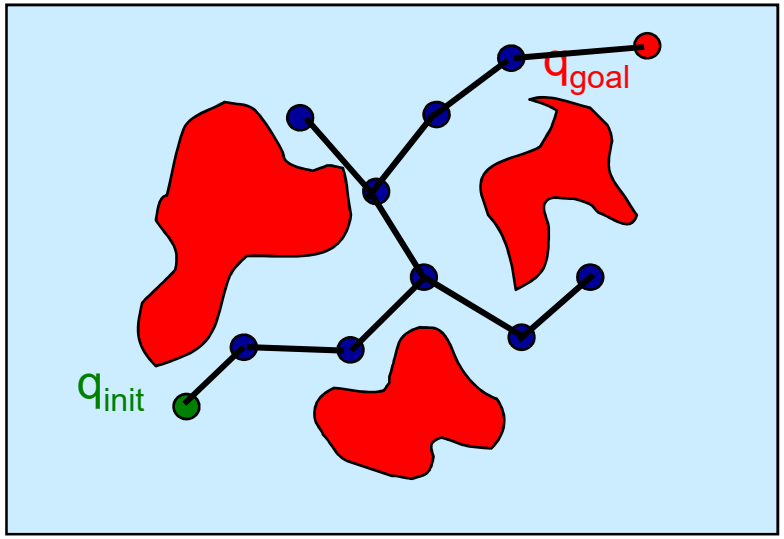
1. Compute q_{rand}
2. Connect to q_{near}
3. Insert q_{new}
4. Go back to 1



RRT: Rapidly-exploring Random trees

- An iterative algorithm

1. Compute q_{rand}
2. Connect to q_{near}
3. Insert q_{new}
4. Go back to 1 until connection between q_{new} and q_{goal}



Exemple
RRT + IK + Motion capture

Problem extensions

- Mobile obstacles
- Several moving objects
- Movable objects
- Deformable objects
- Partial knowledge of the environment
- Uncertainty in perception and control
- Non-holonomic constraints
- Optimal planning
- Visibility constraints

Conclusion

- A large set of techniques have been proposed in motion planning
  - Deterministic search
    - » Approximative representation
    - » Exact representation
  - Potential fields
  - Probabilistic sampling

- Choice relies on
  - Needed performance
  - Problem dimensions
  - Need of completeness

Conclusion

- In practice
  - Planning in high dimensional space
    - » Possible in theory
    - » But: memory consumption, computation time…
  - First thing to do: reduce problem dimensionality
    - » Example: pedestrians
      - Approximation with a cylinder
      - 2D C-space
      - Separation between planning / navigation and animation

Search algorithms

Graphs and path planning

- All planning problems can be repesented by a graph
  - Nodes are configurations
  - Edges are paths between configurations

- *Solving a planning problem is equivalent to finding a sequence of nodes and edges in a graph from one source node to a destination node*

- *Generally solved by finding the shortest path between a source node and a destination node*
  - *A cost is associated to each edge of the graph*

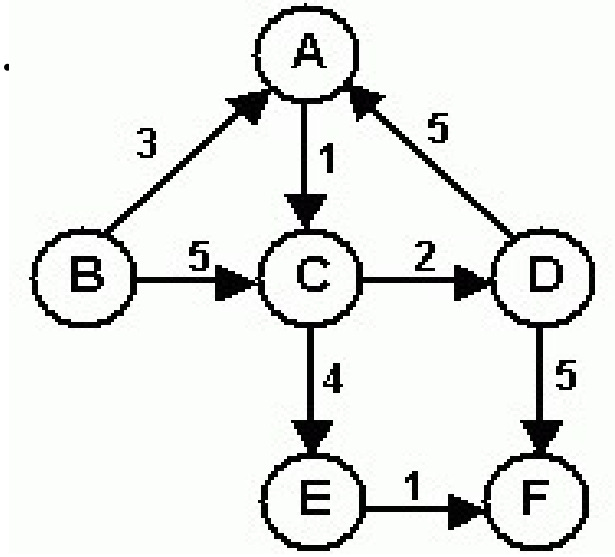
What is the shortest path problem?

- In an edge-weighted graph, the weight of an edge measures the cost of traveling that edge.
- For example, in a graph representing a network of airports, the weights could represent: distance, cost or time.
- Such a graph could be used to answer any of the following:
  - What is the fastest way to get from A to B?
  - Which route from A to B is the least expensive?
  - What is the shortest possible distance from A to B?
- Each of these questions is an instance of the same problem:

The shortest path problem!

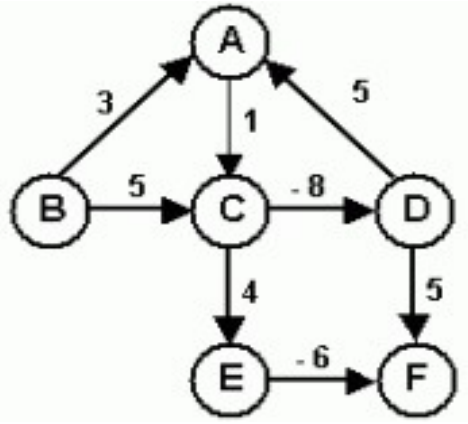
Is the shortest path problem well defined?

- If all the edges in a graph have non-negative weights, then it is possible to find the shortest path from any two vertices.
- For example, in the figure below, the shortest path from B to F is { B, A, C, E, F } with a total cost of 9.
- Thus, the problem is well defined for a graph that contains non-negative weights.



Is the shortest path problem well defined?

- Things get difficult for a graph with negative weights.
- For example, the path D, A, C, E, F costs 4 even though the edge (D, A) costs 5 -- the longer the less costly.
- The problem gets even worse if the graph has a negative cost cycle. e.g. {D, A, C, D}
- A solution can be found even for negative-weight graphs but not for graphs involving negative cost cycles.



{D, A, C, D, A, C, E, F} = 2

{D, A, C, D, A, C, D, A, C, E, F} = 0

The Dijsktra's algorithm

- This algorithm solves the single source shortest path problem for a graph with non-negative weights
- **It finds the shortest path from an initial vertex to all the other vertices**

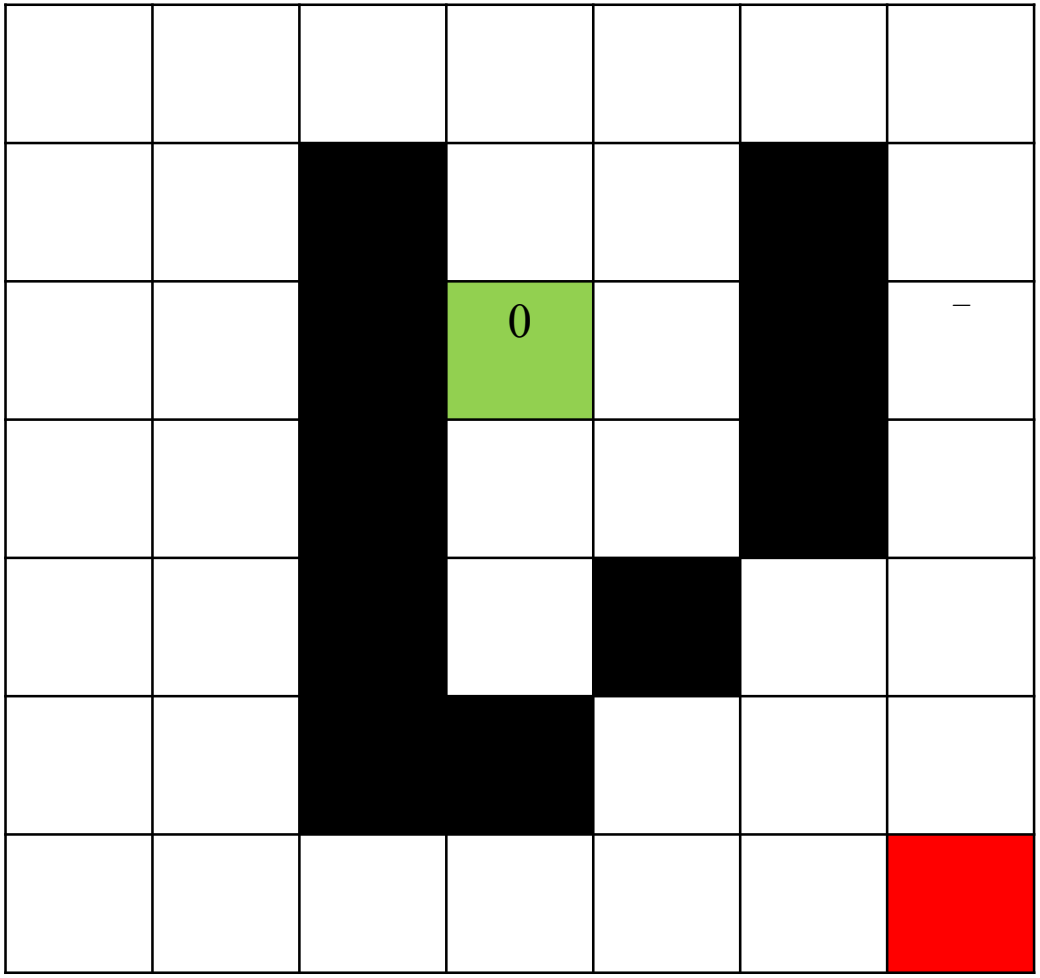
Dijsktra's algorithm: notations

- **Q** is a priority queue containing pairs (cost, node)
  - This queue is sorted in increasing order of cost
- **cost(a,b)** is a function returning the cost associated to the edge linking nodes **a** and **b**
- **cost[a]** is the cost of the optimal path linking **a** to the source node, default value is *infinity*
- **pre[a]** is the predecessor of **a** on the path to the source node, default value is undefined

Dijsktra's algorithm

```
Q.push(0, source)
while(!Q.empty()) {
    c = Q.pop()
    for all successors S of c {
        if(cost(c,S)+cost[c]<cost[S]) {
            Q.push(cost(c,S)+cost[c], S)
            cost[S] = cost[c] + cost(c,S)
            pre[S]=c
        }
    }
}
```

Example



Example

5	4	3	2	3	4	5
6	5		1	2		6
7	6		0	1		7
8	7		1	2		8
9	8		2		10	9
10	9				11	10
11	10	11				11

The A* algorithm

- The A* algorithm computes a shortest path from a source vertex to a given destination vertex
  - Faster than Dijsktra as it expands less nodes
- It uses a heuristic to estimate the cost of reaching the target vertex from the source vertex
- An **admissible heuristic** is a function that exactly or under estimate the cost of reaching the target from the node
  - *Garanties that the A* algorithm will find the shortest path*

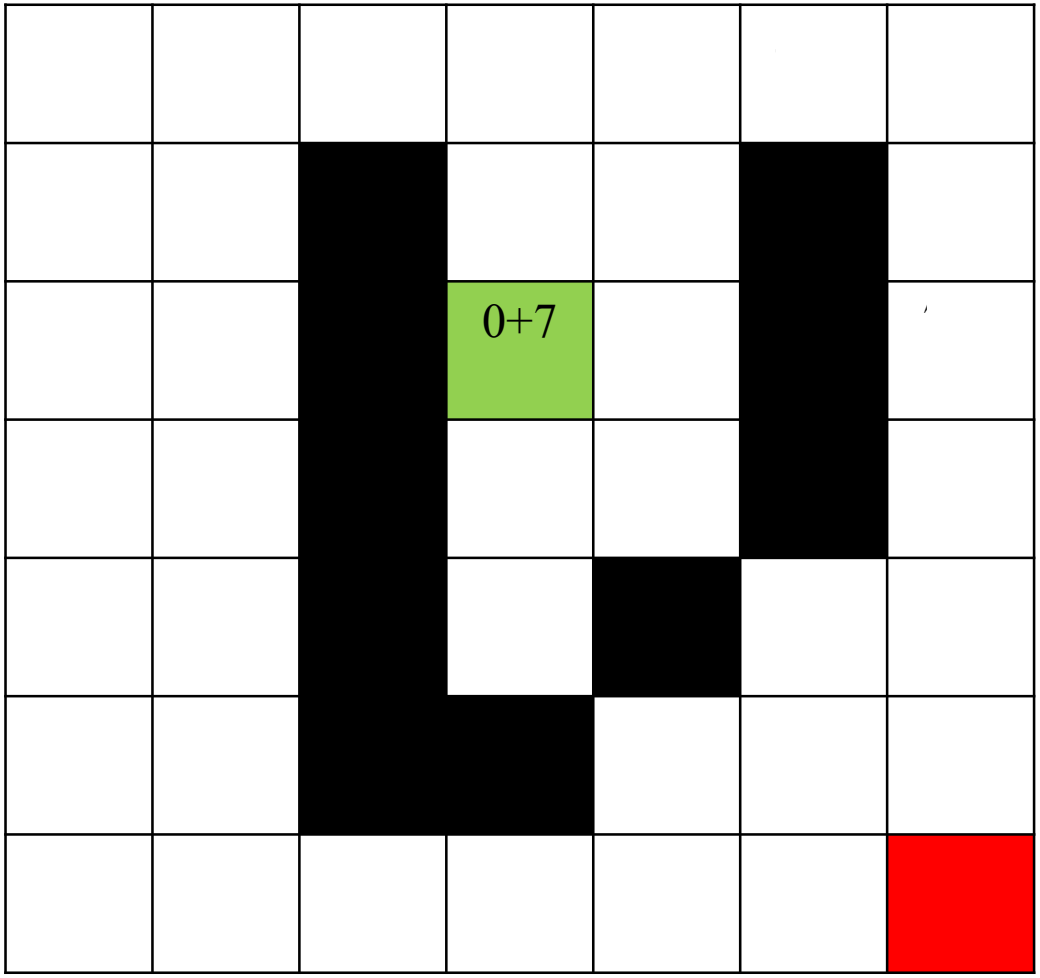
The A* algorithm: notations

- **Q** is a priority queue containing pairs (cost, node)
  - This queue is sorted in increasing order of cost
- **cost(a,b)** is a function returning the cost associated to the edge linking nodes a and b
- **cost[a]** is the cost of the optimal path linking a to the source node, default value is *infinity*
- **pre[a]** is the predecessor of a on the path to the source node, default value is undefined
- **H(a)** is the heuristic function evaluating the cost from vertex a to the destination vertex

The A* algorithm

```
Q.push(H(source), source)
while(!Q.empty()) {
    c = Q.pop()
    if(c==target) { /* Path is found ! */ }
    for all successors S of c {
        if (cost(c,S)+cost[c]<cost[S]) {
            Q.push(cost(c,S)+cost[c]+H(S), S)
            cost[S] = cost[c] + cost(c,S)
            pre[S]=c
        }
} }
```

Example



Example

			2+9	3+8	4+7	5+6
			1+8	2+7		6+5
			0+7	1+6		7+4
			1+6	2+5		8+3
			2+5		10+3	9+2
					11+2	10+1
						11+0

Comparison between Dijsktra and A*

5	4	3	2	3	4	5
6	5		1	2		6
7	6		0	1		7
8	7		1	2		8
9	8		2		10	9
10	9				11	10
11	10	11				11

			2+9	3+8	4+7	5+6
			1+8	2+7		6+5
			0+7	1+6		7+4
			1+6	2+5		8+3
			2+5		10+3	9+2
					11+2	10+1
						11+0

The weighted A*

- Let define H'(n) = Ɛ H(N) with Ɛ > 1
- If H'(n) is used as a heuristic in the A*
  - Produced path is suboptimal and is no more than Ɛ times longer than the optimal path
  - Can speed up the algorithm by exploring less nodes

Impact of the heuristic function

- If H(n)=0, A* turns into Dijsktra algorithm
- If H(n) is always lower or equal to the cost of moving from n to the target, A* is guaranteed to find the shortest path
- If H(n) is always equal to the cost of moving from n to the target, A* exactly follows the shortest path
- If H(n) is sometimes greater than the cost of moving from n to the target, A* is not guaranteed to find a shortest path but can run faster

The IDA* algorithm

- The IDA* algorithm computes a shortest path from a source vertex to a given destination vertex
- IDA* is a variant of the A* search which uses iterative deepening to keep memory usage lower than A*
- Suitable for the exploration of very large graphs that do not fit into memory
  - Ex : implicit graphs generated by the exploration of possible states

The IDA* algorithm

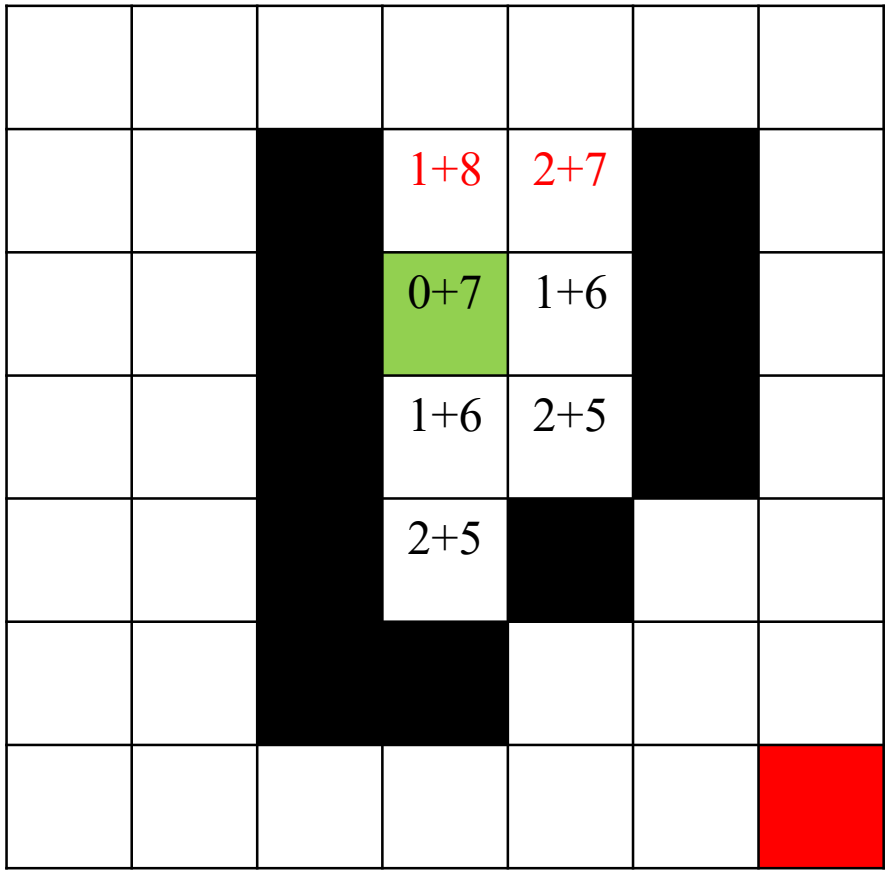
```
IDA_Star(source, target) {
    cost_limit = H(source)
    solution = None
    while(true) {
        (solution, cost_limit) = DFS(0, source, cost_limit, [source])
        if(solution!=None) { return (solution,  cost_limit) } /* Solution found */
        if(cost_limit==infinity) { return None } /* No solution exist */
    }
}
```

The IDA* algorithm

```
DFS(start_cost, node, cost_limit, path) {
    mini_cost = start_cost + H(node)
    if(mini_cost>cost_limit) { return (None, mini_cost) }
    if(node==target) { return (path, cost_limit) }
    next_cost_limit = infinity
    for all successors S of node {
        newStartCost = start_cost+cost(node, S)
        (solution, new_cost_limit) = DFS(newStartCost, S, cost_limit, path+[S])
        if(solution!=None) { return (solution, new_cost_limit) }
        next_cost_limit = min(next_cost_limit, new_cost_limit)
    }
    return (None, next_cost_limit)
}
```

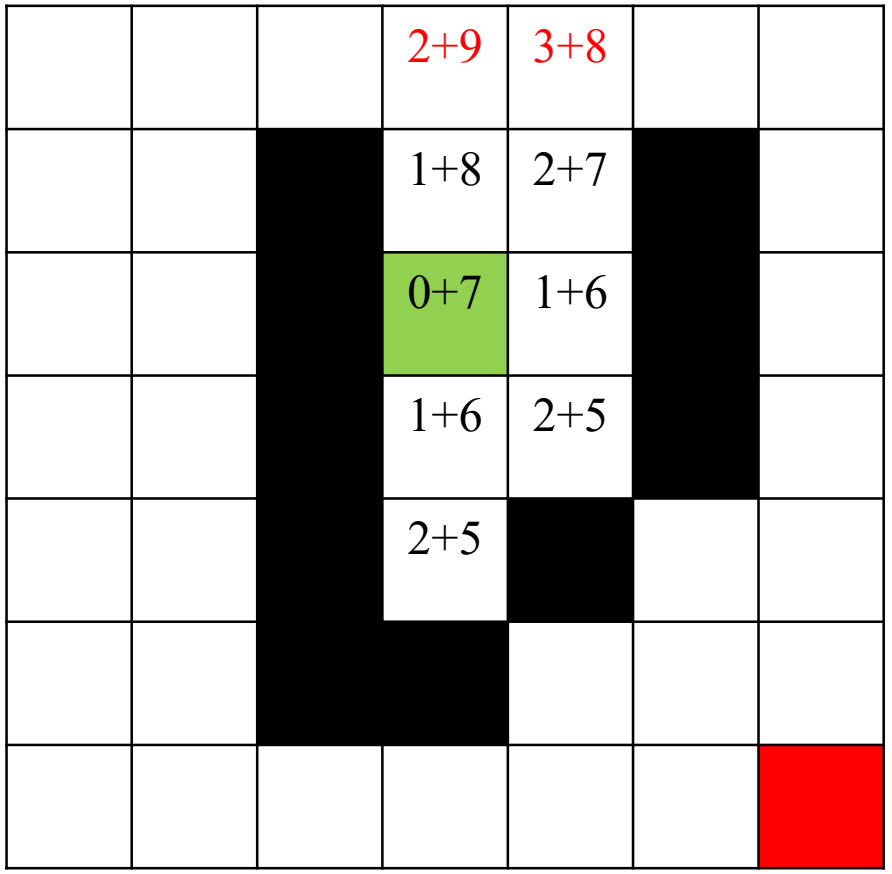
Example

- Pass 1: cost_limit = 7
  - new limit = 9



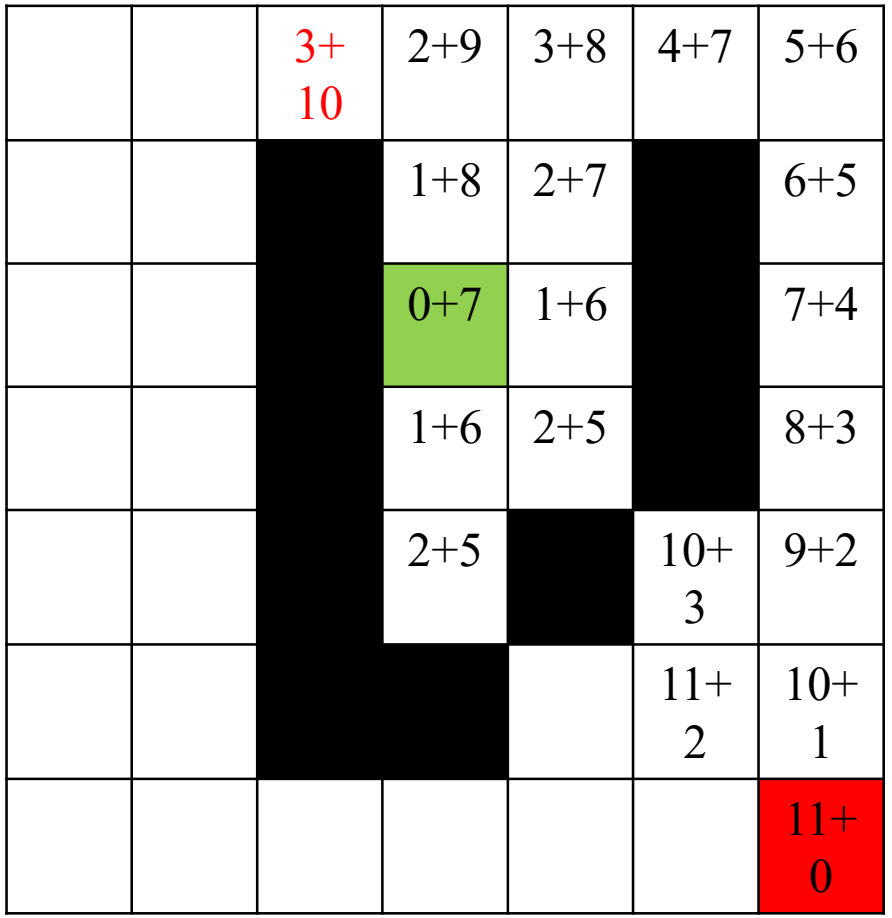
Example

- Pass 1: cost_limit = 7
  - new limit = 9
- Pass 2 : cost_limit = 9
  - new limit = 11



Example

- Pass 1: cost_limit = 7
  - new limit = 9
- Pass 2 : cost_limit = 9
  - new limit = 11
- Pass 3: cost_limit = 11
  - Goal reached



IDA* algorithm

- Without enhancements
  - Slower than A*
    - » A* uses a sorted Q => impact on performances
    - » IDA* explores several times the same configurations with different paths
- Possible enhancements
  - Do not explore nodes already present in the current path
  - Using a cache to store best estimated path lengths in some configurations

Heuristics in path planning

- In grids
  - Manhattan distance
    - » Useful in case of 4-connexity
    - » h(n) = D*(abs(n.x-goal.x) + abs(n.y-goal.y))
    - » D: min cost of a move to an adjacent cell
  - Diagonal distance V1
    - » Useful in case of 8-connexity
    - » h(n) = D * max(abs(n.x-goal.x), abs(n.y-goal.y))
    - » D: min cost of a move to an adjacent cell

Heuristics in path planning

- In grids
  - Diagonal distance V2
    - » h_diagonal(n) = min(abs(n.x-goal.x), abs(n.y-goal.y))
    - » h_straight(n) = (abs(n.x-goal.x) + abs(n.y-goal.y))
    - » h(n) = D2 * h_diagonal(n) + D * (h_straight(n) - 2*h_diagonal(n)))
    - » D2 : cost of a diagonal move
    - » D : cost of a vertical / horizontal move
- In continuous space / grids
  - Euclidian distance
    - » h(n) = D * sqrt((n.x-goal.x)^2 + (n.y-goal.y)^2)

Which algorithm for what purpose?

- Dijsktra
  - Useful to compute all shortest paths to a given target from all nodes
- A*
  - Useful to compute the shortest path from a source to a destination
- IDA*
  - Useful to compute the shortest path from a given source to a given target if graph is implicitly generated and cannot be stored explicitly

Discussions on examples

- Which algorithm and (eventually) which heuristic?

- A set of robots should reach a unique target
  - Environment is represented with a grid
- Flee from a set of dangerous positions
  - Environment is represented with a roadmap
- Find the nearest cash dispenser knowing several cash dispensers positions in a 2D world
  - Environment is represented with a roadmap

Other kind of algorithms

- Incremental replanning algorithms
  - Handle changes in costs associated to the graph
  - Ex: D*, focussed D* and D* lite algorithms
- Anytime algorithms
  - Agent must react quickly and the planning problem is complex
  - Find an approximate solution, then refine it
  - ARA* (Anytime repairing A*)

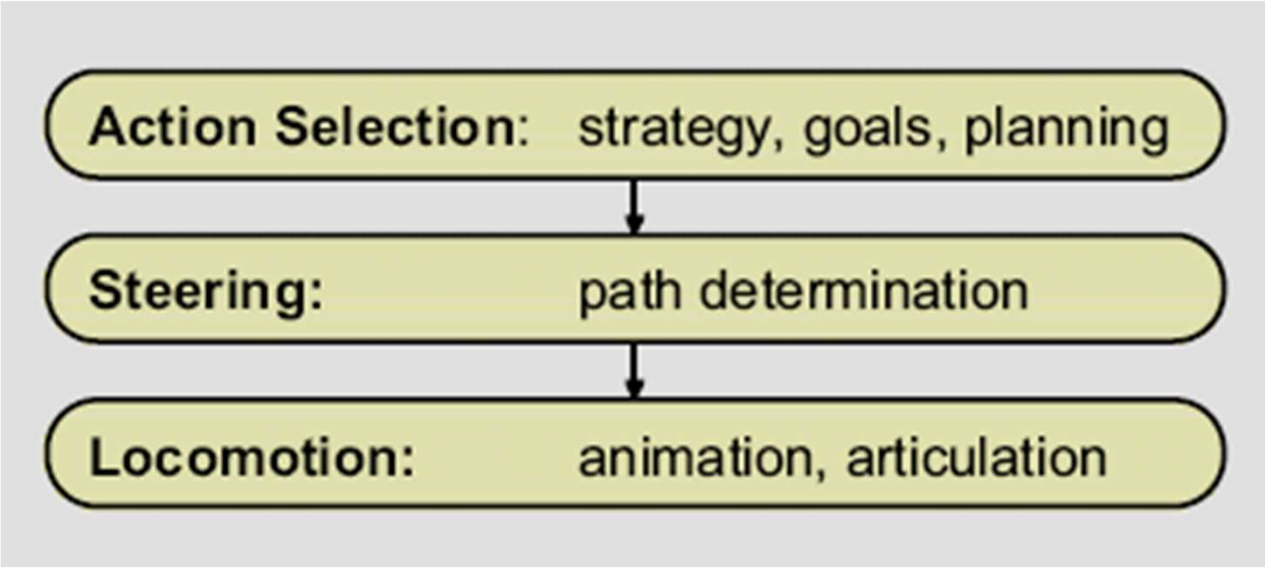
- Dave Ferguson, Maxim Likhachev, and Anthony Stentz. A Guide to Heuristic-based Path Planning. ICAPS 2005.
  http://www.ri.cmu.edu/publication_view.html?pub_id=5085

Steering behaviors and group movements

Introduction

- Autonomous Characters
  - Autonomous agents
  - NPCs in Games (Non-player characters)
- Applications
  - Robotics
  - Artificial Intelligence (AI)
  - Artificial Life
- References
  - Craig W. Reynolds
  - 1987 “Flocks, Herds, and Schools: A Distributed Behavioral Model”, Siggraph’87 Proceedings
  - 1999 “Steering Behaviors for Autonomous Characters”, GDC Proceedings*
  - www.red3d.com/cwr/steer

Motion behavior



Steering

- Path Determination
  - Path finding or path planning
- Behaviors
  - Seek & flee
  - Pursuit & evasion
  - Obstacle avoidance
  - Wander
  - Path following
  - Unaligned collision avoidance
- Group steering
  - Separation, cohesion, alignement

A simple vehicle model

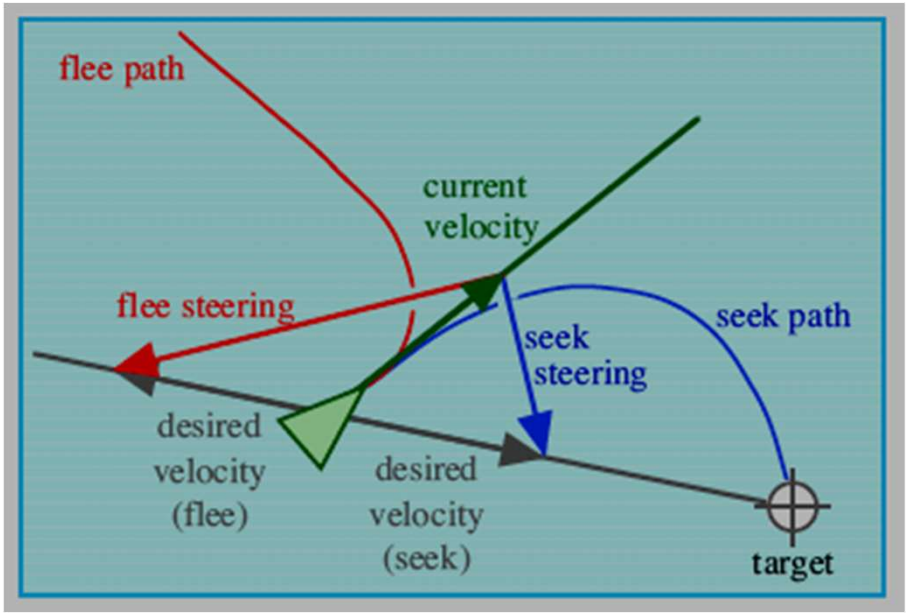
- A Point Mass
  - Linear momentum
  - No rotational momentum
- Parameters
  - Mass (scalar)
  - Position (vector)
  - Velocity (vector)
    - Modified by applied forces
  - Max speed (scalar)
    - Top speed of a vehicle
  - Max steering force (scalar)
    - Self-applied
  - Orientation (scalar)
    - Car / aircraft

Computation of a new position: Euler integration

- steer_force = Truncate(steering, Max_force)
- Acceleration = steer_force / mass
- Velocity = Truncate(Old_velocity + Acceleration * Time_step, Max_speed)
- New_position = Old_position + Velocity * Time_step
  - Steering: difference between current velocity and desired velocity
  - Truncate: a function that limits the norm of the vector to a maximum value
  - Time_step: time elapsed since last update

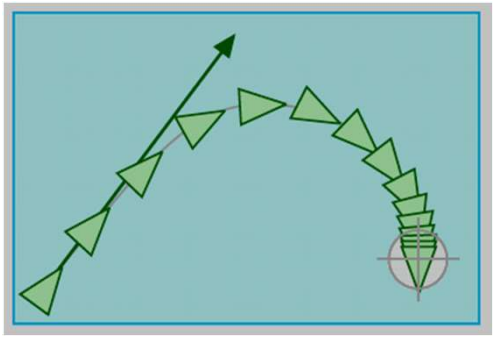
Seek and flee behaviors

- Seek: pursuit to a Static Target
  - Steer a character toward to a target position
- Seek Steering force
  - desired_velocity = normalize(target - position)*max_speed
  - steering = desired_velocity – velocity
- Flee
  - Inverse of Seek
- Variants
  - Arrival
  - Pursuit of a moving target



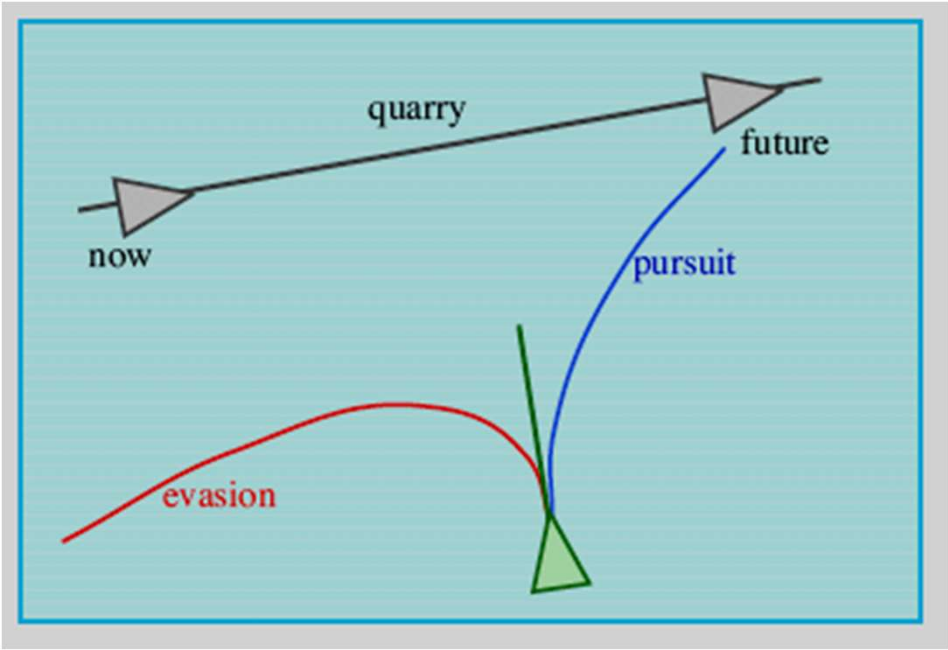
Arrival behavior

- A Stopping Radius
  - Outside the radius, arrival is identical to seek
  - Inside the radius, the speed is decreased down to zero
    - » target_offset = target – position
    - » distance = length(target_offset)
    - » ramped_speed = max_speed*(distance/slowing_distance)
    - » clipped_speed = minimum(ramped_speed, max_speed)
    - » desired_velocity = (clipped_speed/distance)*target_offset
    - » steering = desired_velocity – Velocity
  - *Rq: slowing_distance is a parameter of the behavior*



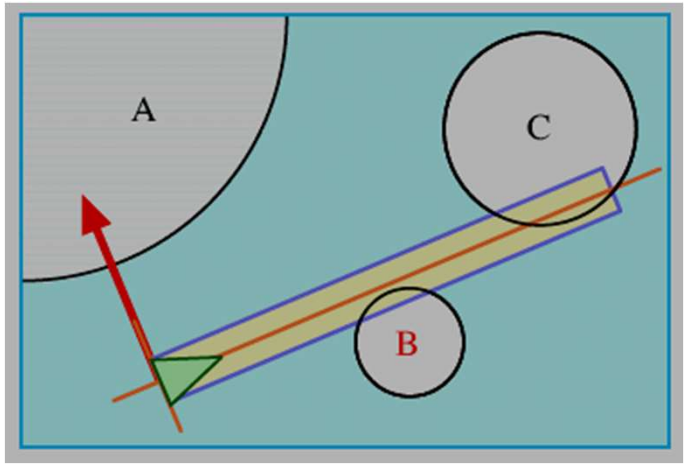
Pursuit and evasion behaviors

- Another character is moving
- Pursuit: apply seek behavior to a character's predicted position
- Evasion: apply flee behavior to a character's predicted position
- Position prediction
  - Ex: linear extrapolation based of oberved velocity vector and a given extrapolation time



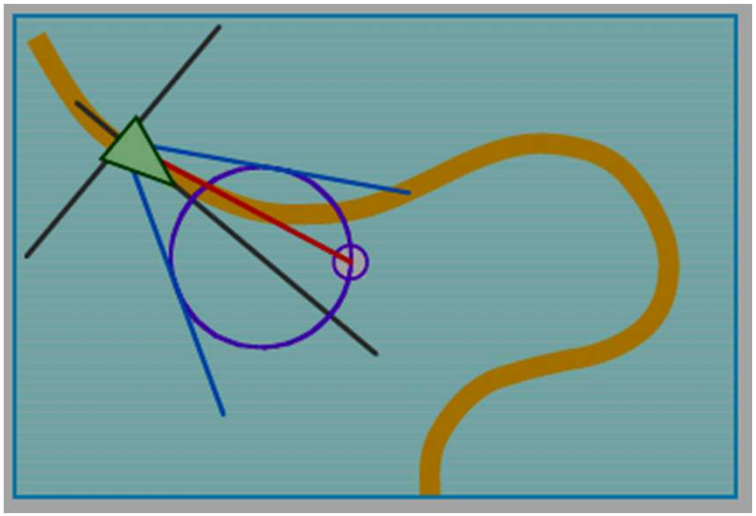
Obstacle avoidance behavior

- Use a bounding sphere / circle for the controlled character
- Obstacle selector
  - All points which distance to a segment is lower than a given threshold
  - Segment: (position, position+velocity*time)
  - Distance threshold: bounding sphere / circle radius



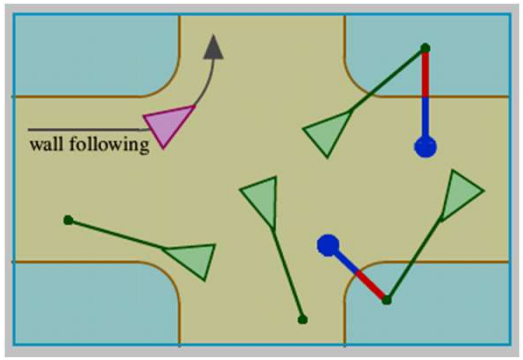
Wander behavior

- Random steering
- One Solution :
  - Retain steering direction state
    - » Constrain steering force to the sphere surface located slightly ahead of the character
  - Make small random displacements of the steering at each frame
    - » A small sphere on sphere surface to indicate and constrain the displacement

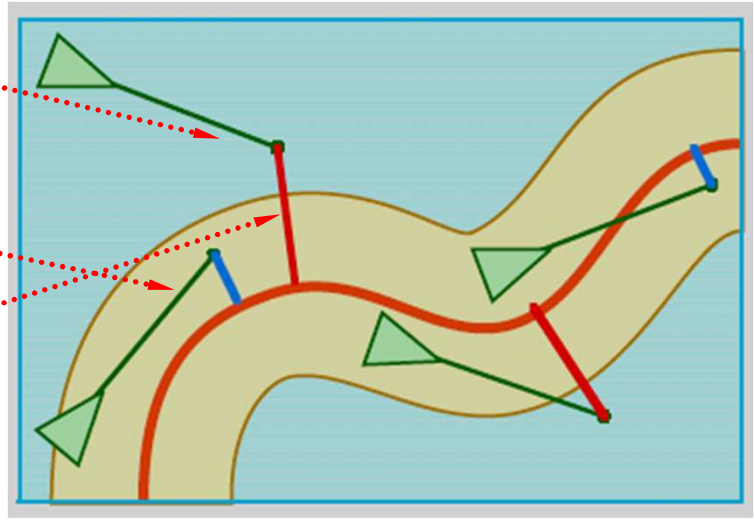


Path following behavior

- A Path
  - Spine
    - » A spline or poly-line to define the path
  - Pipe
    - » The tube or generated cylinder by a defined “radius”

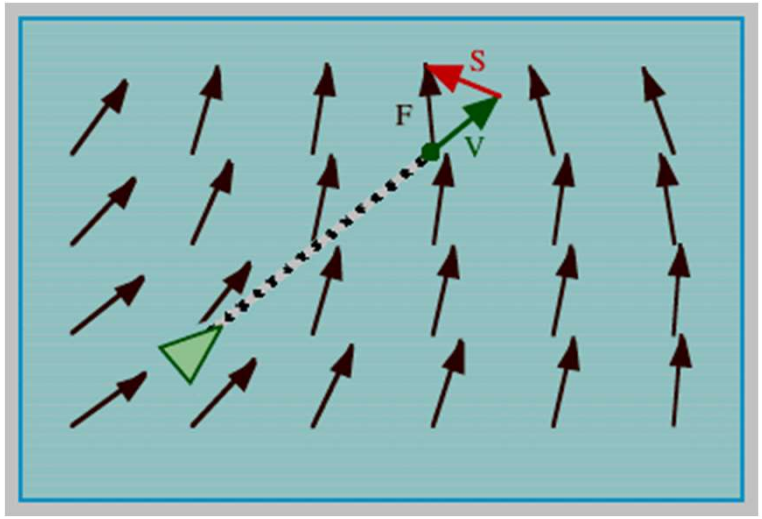


- Following
  - A velocity-based prediction position
    - » Inside the tube
      - Do nothing about steering
    - » Outside the tube
      - “Seek” to the on-path projection



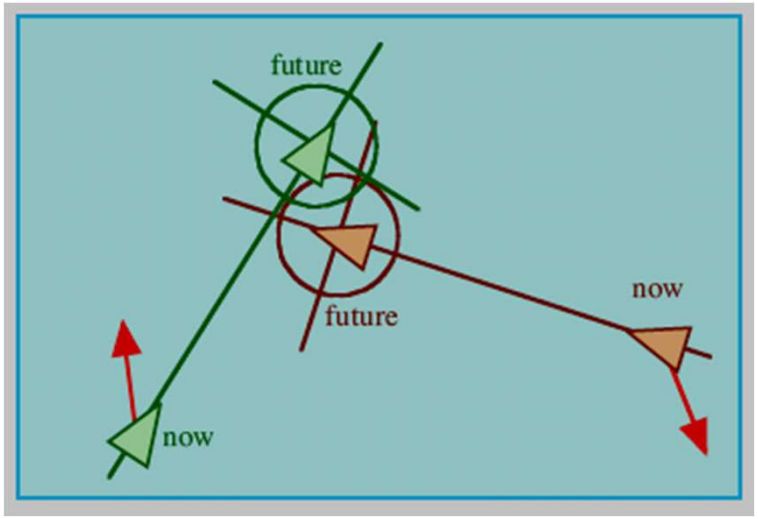
Flow field following behavior

- A Flow Field Environment is Defined
- Computation of the steering force
  - The next position of the character is estimated
  - The flow field is sampled at this position (flow_velocity)
  - Steering = flow_velocity - velocity



Unaligned collision avoidance behavior

- Turn Away from Possible Collision
- Predict the Potential Collision
  - Use bounding spheres
- If possibly collide,
  - Apply the steering on both characters
  - Steering direction is possible collision result
    - » Use “future” possible position
    - » The connected line between the two sphere centers



Steering of groups of characters

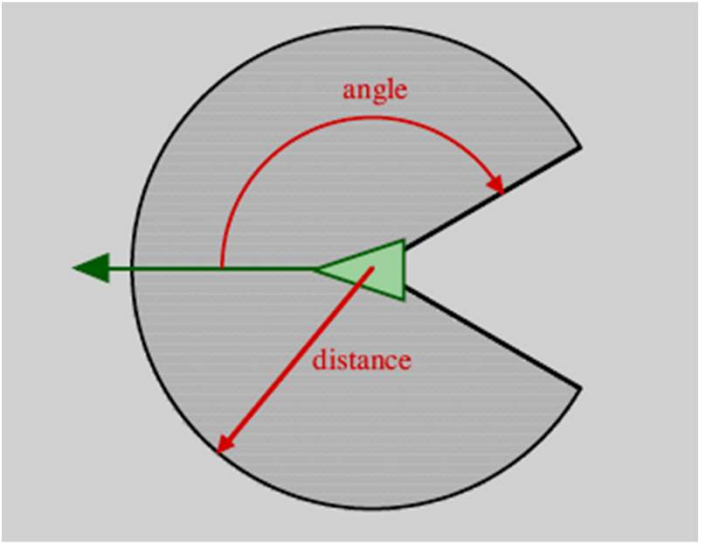
- Steering behaviors determining how the character reacts to the other characters within his local neighborhood

- Behaviors include
  - Separation
  - Cohesion
  - Alignment

Local neighbourhood

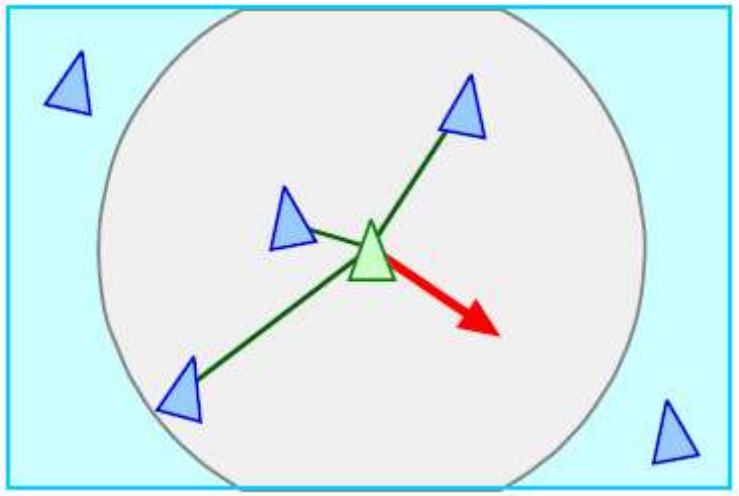
- The local neighborhood is defined with
  - A distance
  - A field-of-view (angle)

- Need of a spatial data structure for neighborhood retrieval
  - Kd-Tree
  - Octree / Quadtree
  - Regular grids
  - …



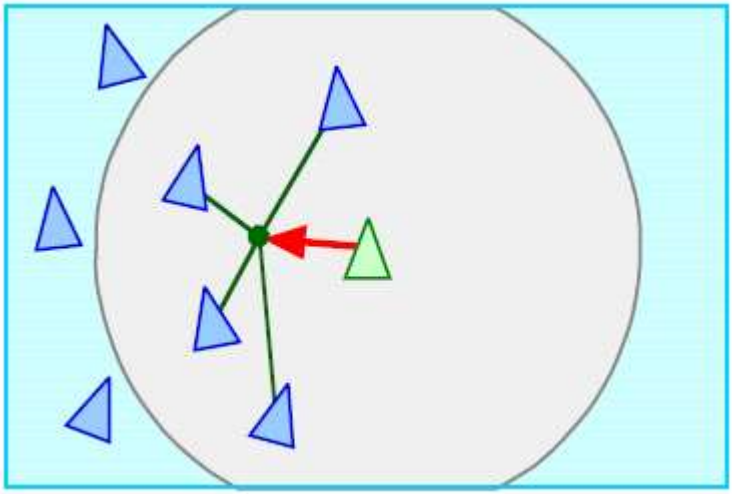
Separation behavior

- The character maintains a distance from others
  - Compute the repulsive forces within local neighborhood
    - » Calculate the position vector for each nearby
    - » Normalize it
    - » Weight the magnitude with distance
      - 1/distance
    - » Sum the result forces
    - » Negate it



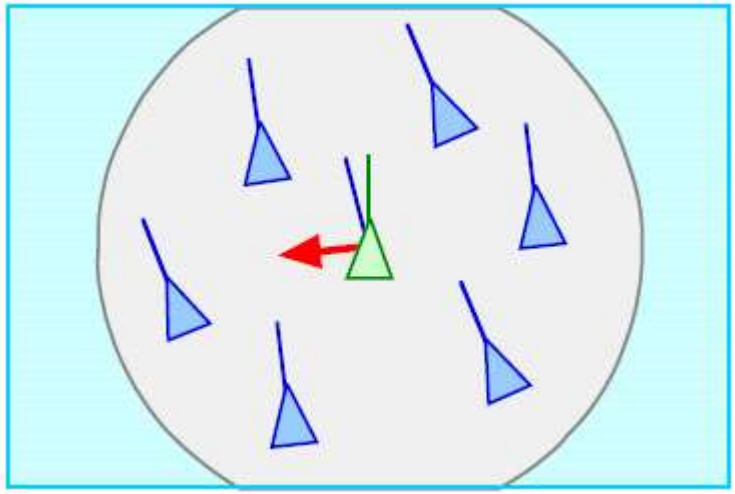
Cohesion behavior

- The character navigates among the others
  - Compute the cohesive forces within local neighborhood
    - » Compute the average position of the other nearbys
      - Gravity center
    - » Apply “Seek” to the position



Alignement behavior

- The character to aligns with the others
  - Compute the steering force
    - » Average the velocity of all other characters nearby
    - » The result is the desired velocity
    - » Correct the current velocity to the desired one with the steering force

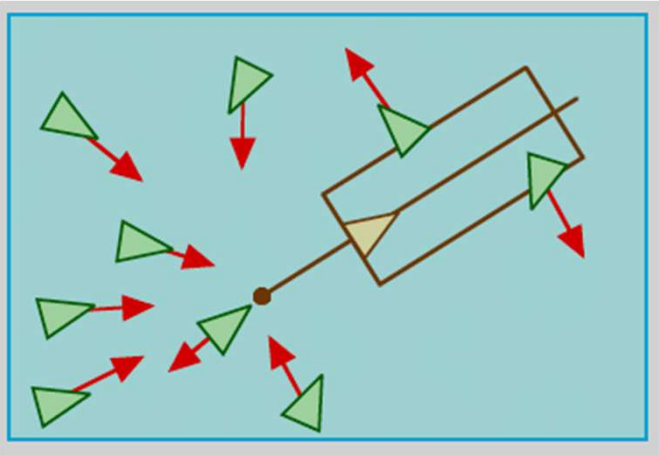


Flocking behaviors

- Boids Model of Flocks
  - [Reynolds 87]
- Combination of
  - Separation steering
  - Cohesion steering
  - Alignment steering
- For Each Combination
  - A weight for combining
  - A distance
  - An Angle

Leader following behavior

- Follow a Leader
  - Stay with the leader
    - » "Pursuit" behavior (Arrival style)
  - Stay out of the leader's way
    - » Defined as "next position" with an extension
    - » "Evasion" behavior when inside the above area
  - "Separation" behavior for the followers



Steering behavior

- A Simple Vehicle Model with Local Neighborhood
- Common Steering Behaviors
  - Seek, Flee, Pursuit, Evasion
  - Offset pursuit, Arrival, Obstacle avoidance
  - Wander, Path following, Wall following
  - Flow field following, Unaligned collision avoidance
  - Separation, Cohesion, Alignment
  - Flocking, Leader following
- Combination of the behaviors defined above
  - Intelligent steering behavior

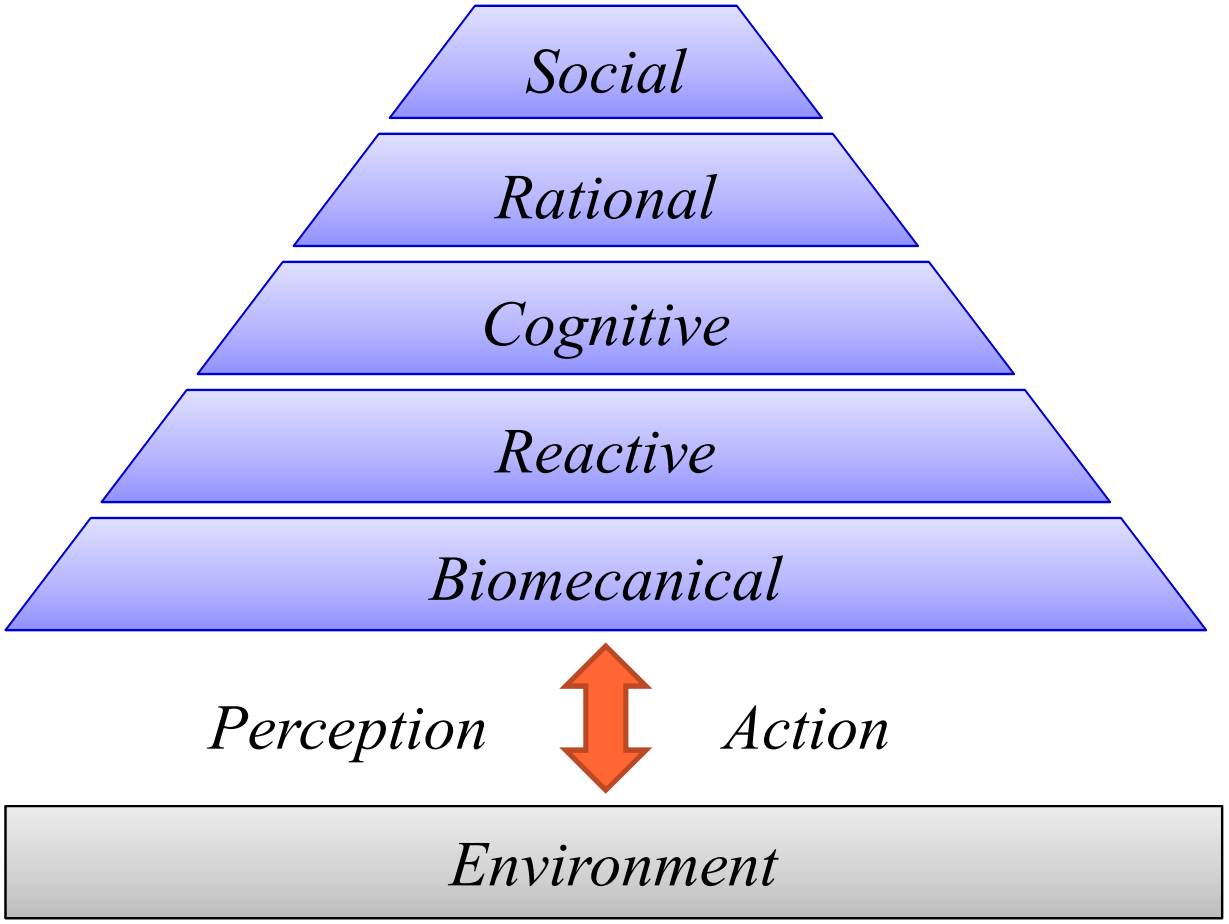
Behavioral animation

Behavior modeling

Behavioral animation

- Goal:
  - Populating virtual environments
  - Interactive applications
    - » Interactions not always predetermined
    - » Automatic adaptation of the characters' behavior
  - Simulation
    - » Crowd simulation: environment validation, security
  - Automating creation processes
    - » Crowds in films
    - » Golaem, RTK Crowds, MASSIVE

Behavior pyramid



Wish list…

Wish list…

The AI should be goal directed

…yet react to sudden changes!

Challenges

- Big environments
- Large number of entities
- Advanced animation & control
- Designer supervision
  - But AI behave autonomously
- AI should be goal directed
  - But react to sudden changes

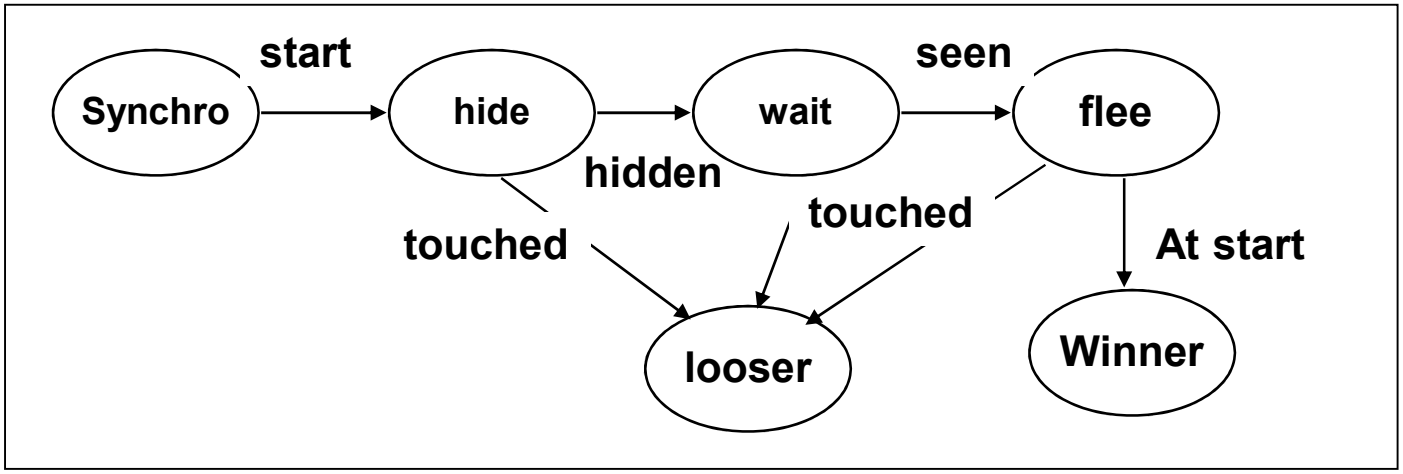
Rule based systems

- General form:
  - If **condition** then **action**
  - Ex :
    - » If near enemy and healthy then attack
    - » If near enemy and not healthy then flee
  - Actions can be atomic or fire a task to complete…

- advantage: simple approach, easy to describe
- Problems:
  - » Conflicts between rules
    - Need of a strategy, in general priorities
  - » Do not ensure temporal consistency of behavior

Finite state machines

- Example: hide-and-seek game



Finite state machines

- Composed of
  - States
    - » An atomic activity
    - » Entry / exit state actions
    - » While in state, activity
  - Transitions
    - » Conditional change of activity
    - » A condition
    - » Possible action on transition
- State machine = logical suit of activities
- One initial state, Several ending state

Finite states machines

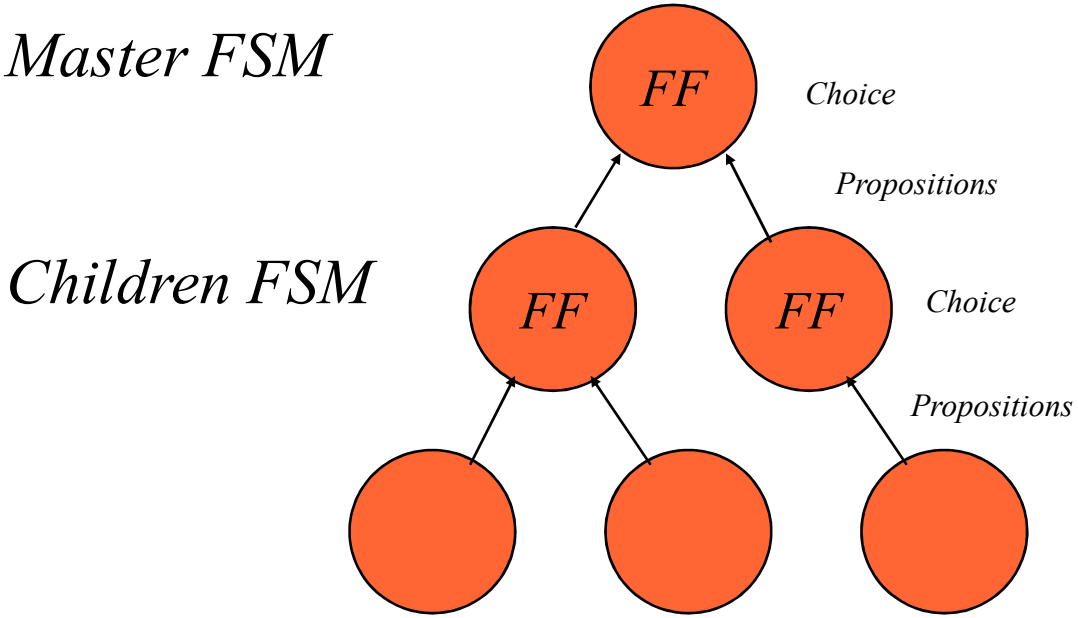
- State machine stacks
  - One state machine can call another
  - Current state machine is pushed, the other is ran
  - Once the other finishes, the calling state machine is pop and continue its execution
  - Transposition of the notion of procedure call for state machines
    - » A state in the upper level is described through a state machine at the lower level

Finite state machines

- Parallel state machines
  - Several state machines can run in parallel
  - Use semaphores and message passing for synchronization and communication
    - » Warning: dead locks…
  - Advantage: simulate the product of the two state machines
  - Example: one state machine for displacement, another to shoot

Finite state machines

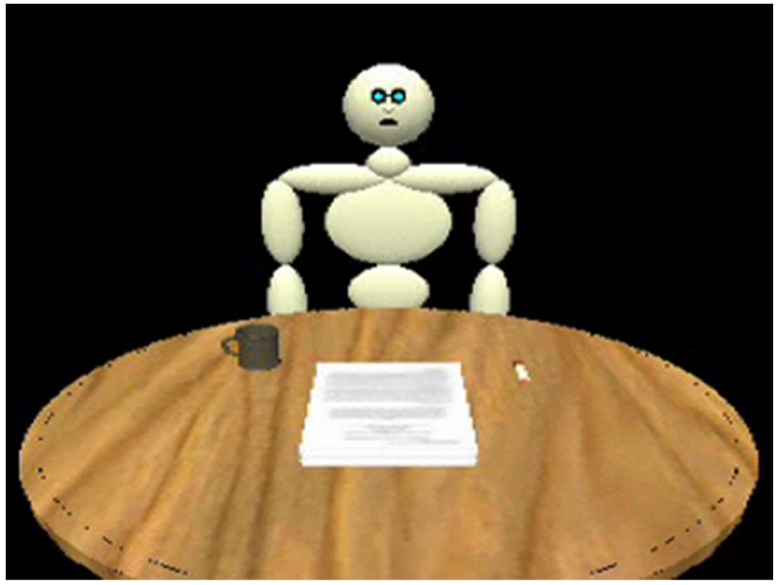
- Hierarchical and parallel state machines
  - Hierarchy = filtering functions
  - Hierarchy = composition of parallel state machines



Finite state machines

- Can be used for behavior and scenarios descriptions
- Pros
  - Simplicity of description
  - Consistency ensured by state machine structure
  - Perfect control
  - Possibility to add time information etc…
- Cons
  - Difficulty to avoid dead locks in case of parallel state machines
  - Hard to maintain when FSM become very complex

Finite state machines



Automatic scheduling of parallel FSM

Dead lock avoidance, priorities, preferences

F Lamarche, and S Donikian (2002). *Automatic Orchestration of Behaviours through the management of Resources and Priority Levels*. In: Autonomous Agents and Multi Agent Systems. ACM, Bologna, Italy

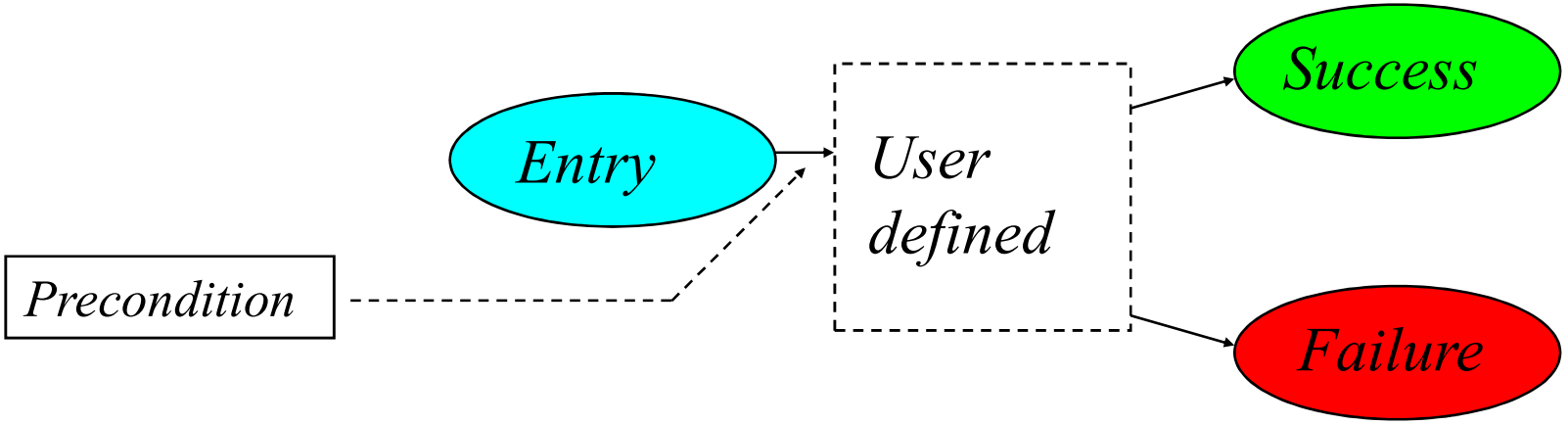
FSM and task modelling

- State machines
  - Useful to describe atomic behaviours
  - Needs a higher description level to describe complex behaviours

- Tasks
  - Define a modelling framework
    - » Provide an abstract behaviour with identified states
  - Uses inheritance and polymorphism
    - » behaviour implementation
  - Operator definition
    - » High level task creation and manipulation

FSM and task modelling

- Task model
  - Three predefined states : Entry / Success / Failure
  - Precondition(s)
  - Conserved resources
  - User defined behaviour through inheritance



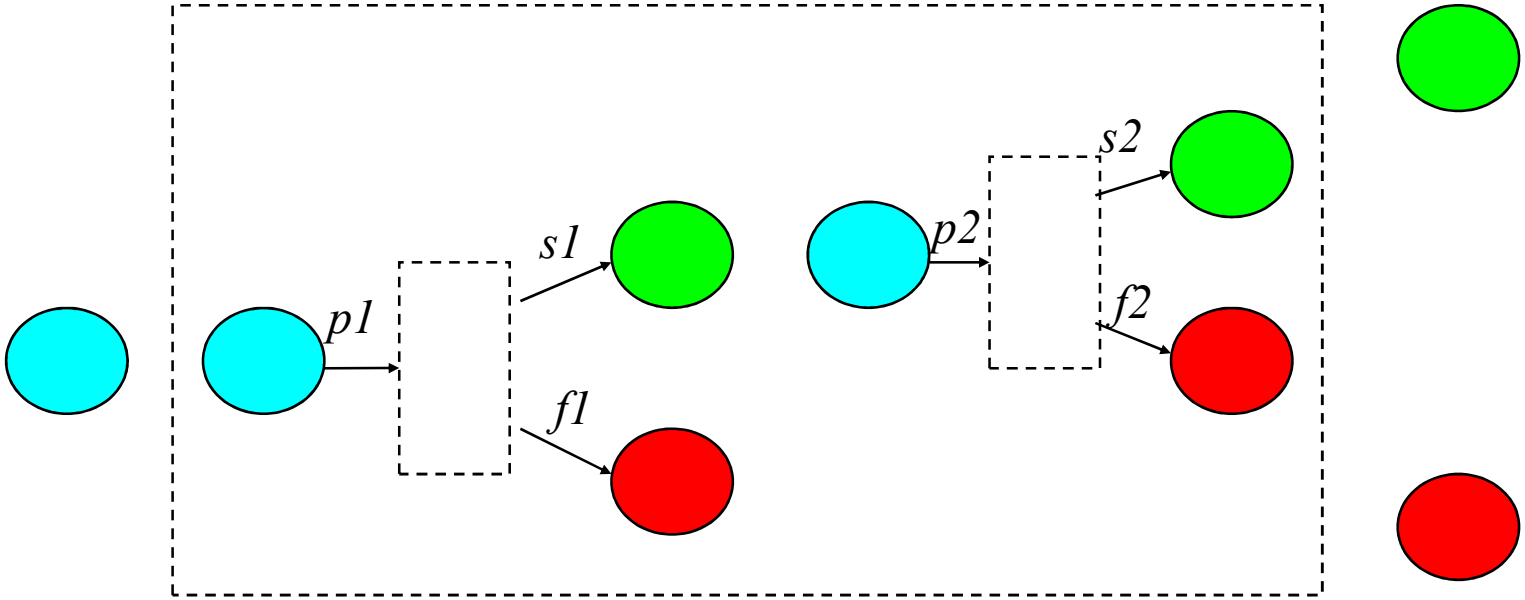
FSM and task modelling

- Task model provides operators
  - Logical combination of different tasks
  - Sequence / on failure / choice / all without order / parallel / repeat until / realize X among all
  - Parameters : tasks to combine
  - State machine generation with structure modification

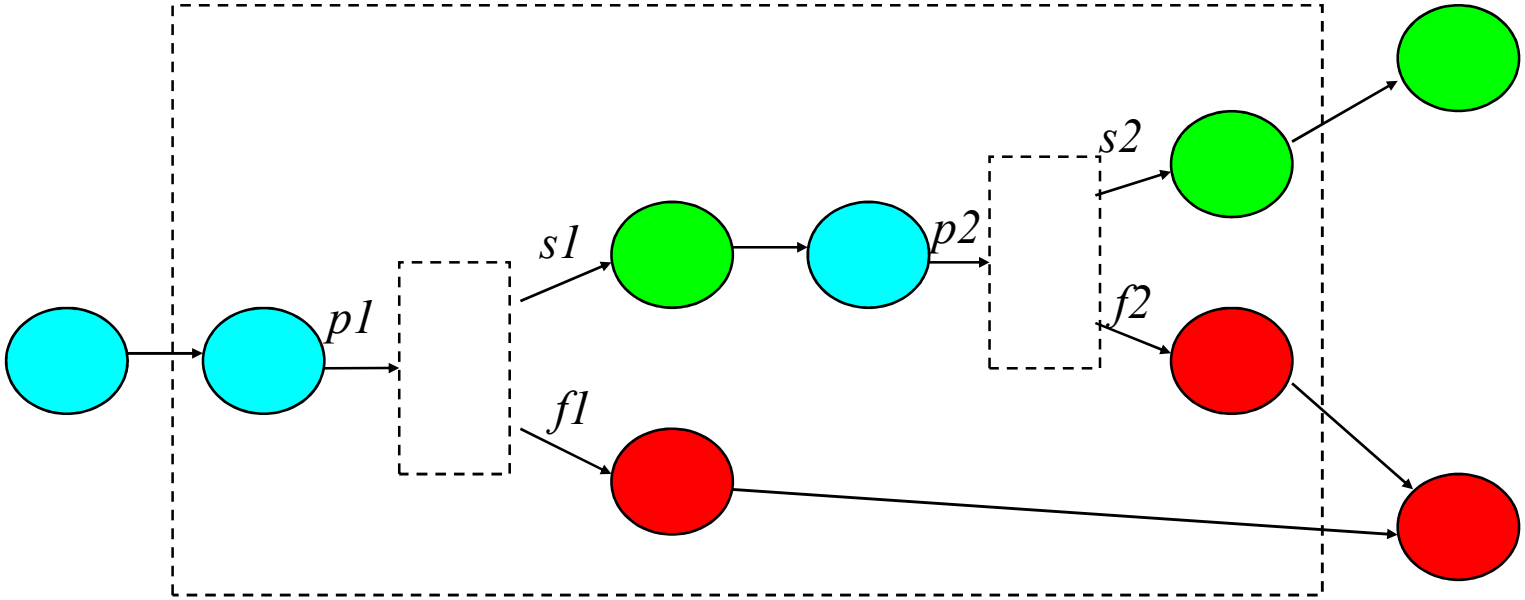
- New task generation
  - Can be combined with other tasks thanks to operators

Example : Sequence operator

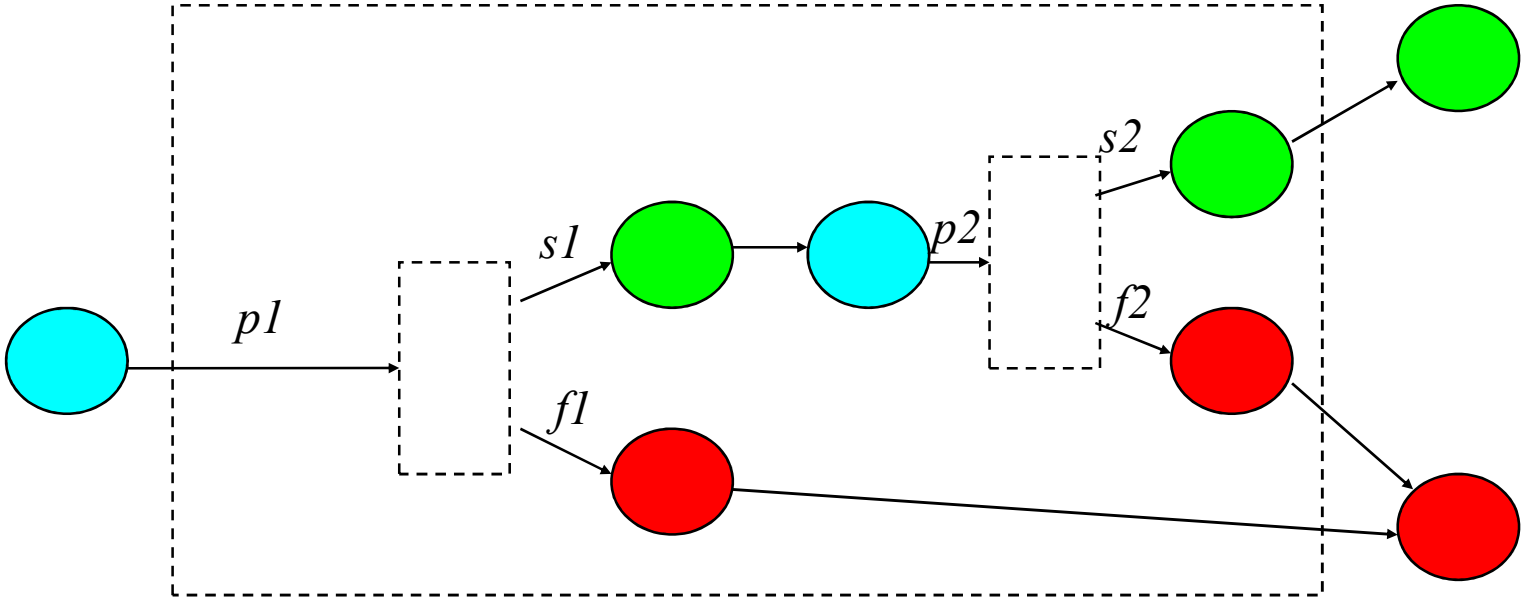
Example : Sequence operator



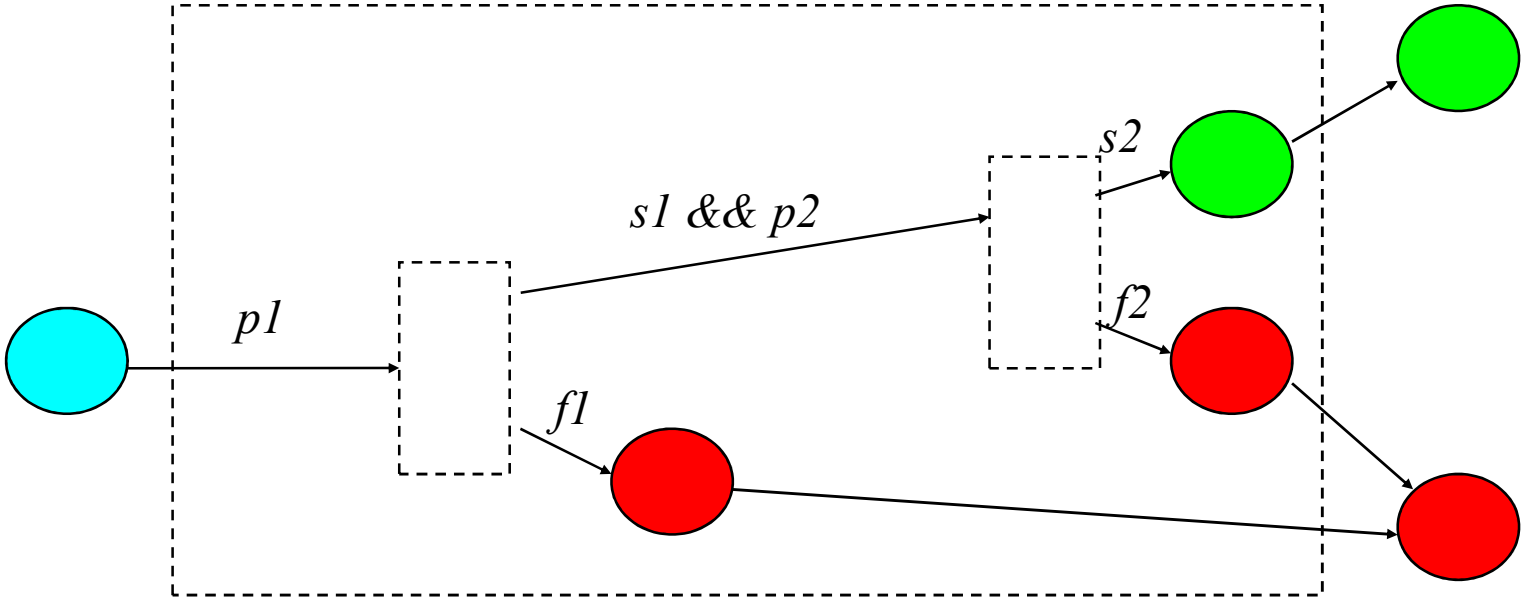
Example : Sequence operator



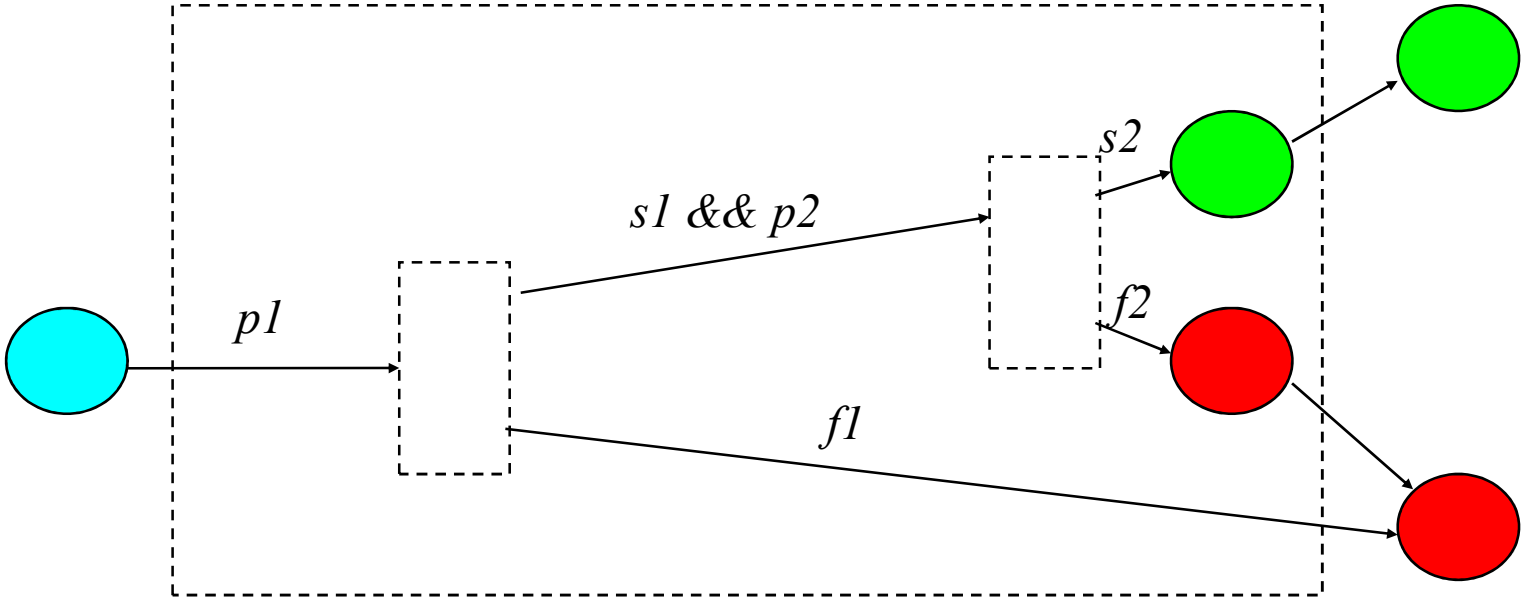
Example : Sequence operator



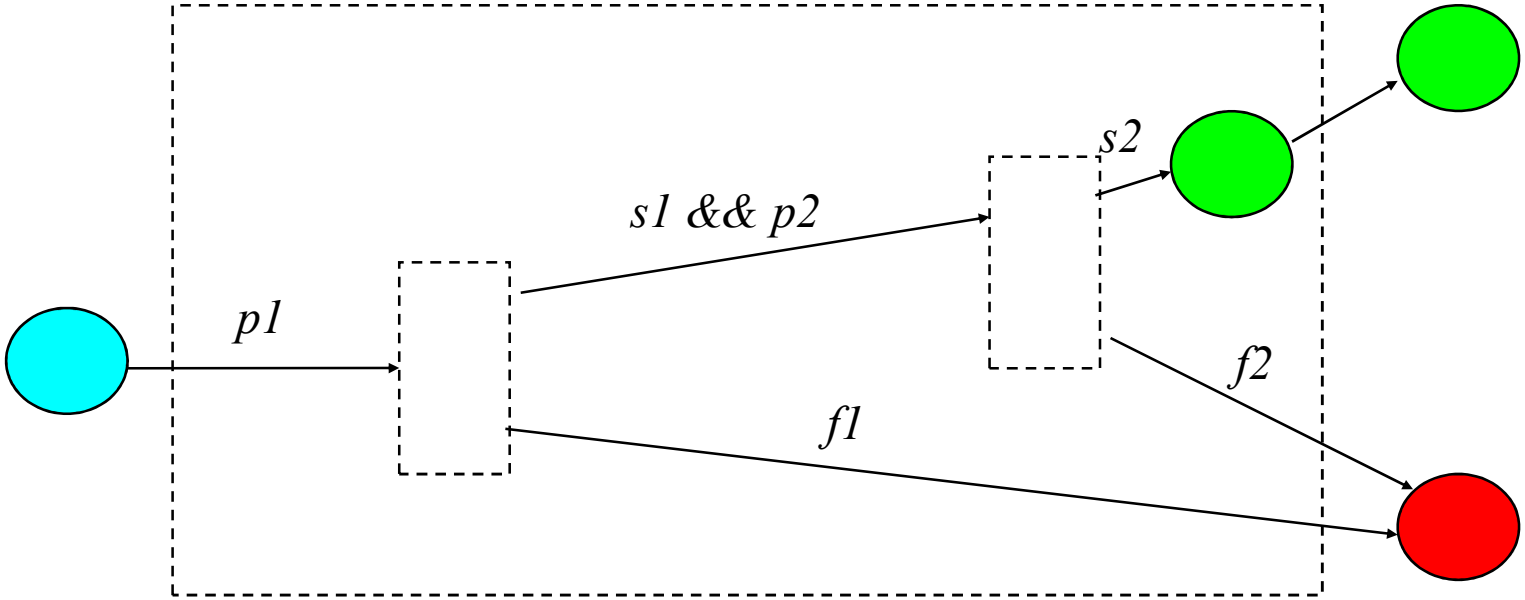
Example : Sequence operator



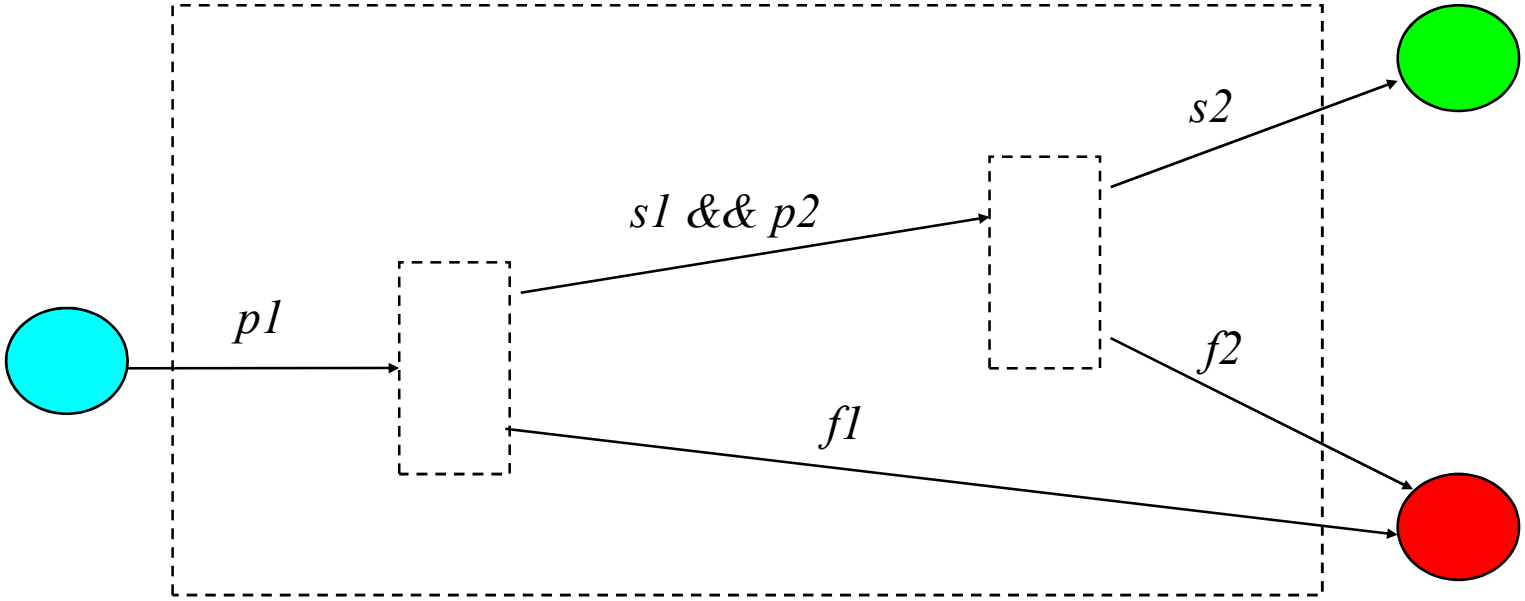
Example : Sequence operator



Example : Sequence operator



Example : Sequence operator



New task generation

FSM and task modeling

- FSM & Task modeling
  - Generic framework providing high level operators
  - Simple description of complex behaviours
  - Easier to maintain than very complex hand designed state machines
  - Reuse of pre-described state machines
    - » Modifications are automatically taken into account
- Remark
  - Keep operators, encode them and replace FSMs by scripts / code and…
    - » **You obtain behavior trees**

Hierarchical task networks

- Classical planning:
  - Each state of the world is represented by a set of atoms
    - » Can be true or false / present or absent
    - » Relate to object state or relations between objects
    - » Ex: (closed door), (on-table my-computer), (at-location me classroom)
  - Each action (called operator) corresponds to a deterministic state transition
    - » Parameters: objects used in preconditions and effects
    - » Precondition: conjunction of atoms that must be present to execute the action
    - » Effect: adds atoms (add list) and remove some (del list)

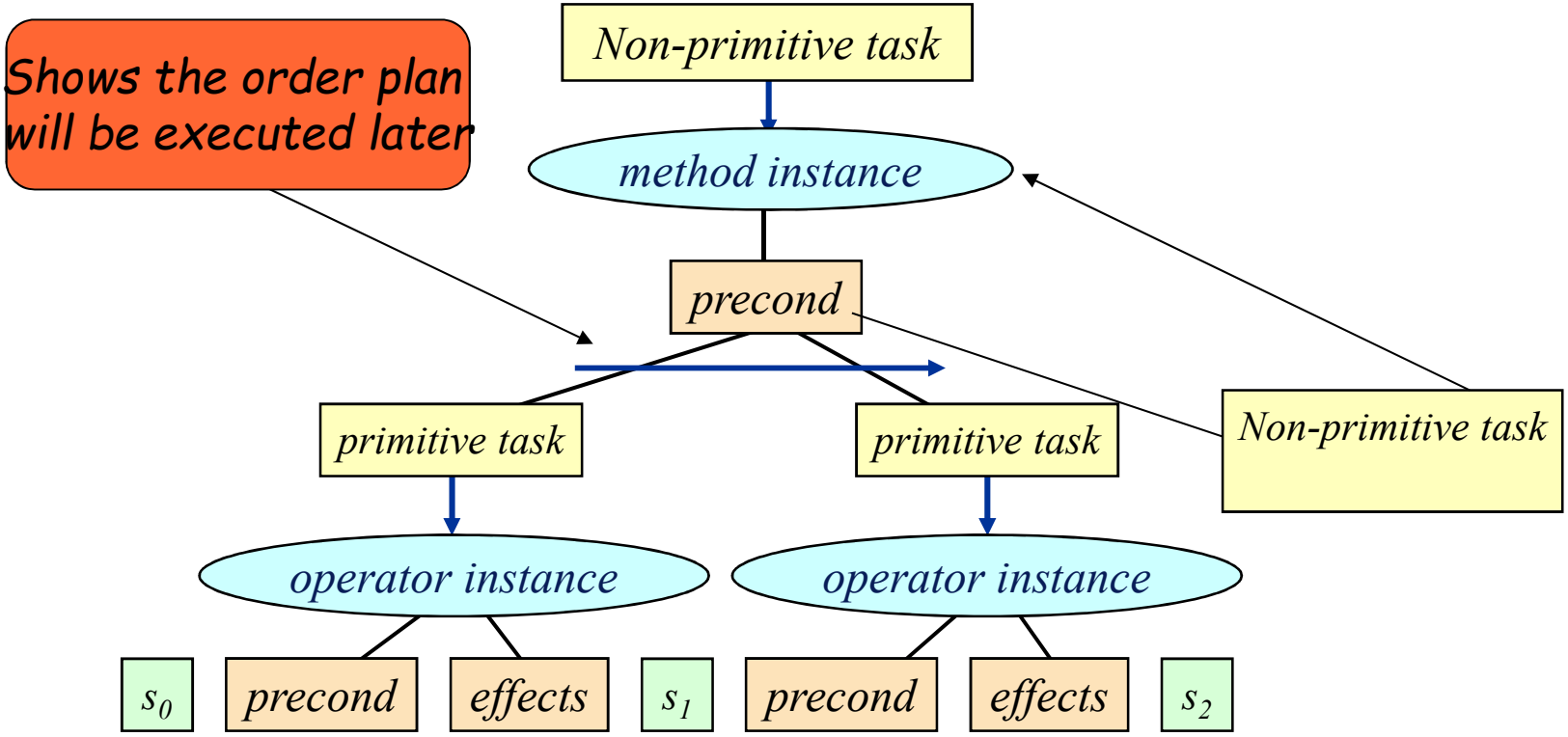
Hierarchical task networks

- Classical planning:
  - Goal : a sub state of the world
  - Ex : (in-corridor me)
  - Search for a succession of actions satisfying the goal

- But
  - We can't describe activities that cannot be expressed as a sub state of the world
    - » Ex: go for a walk…

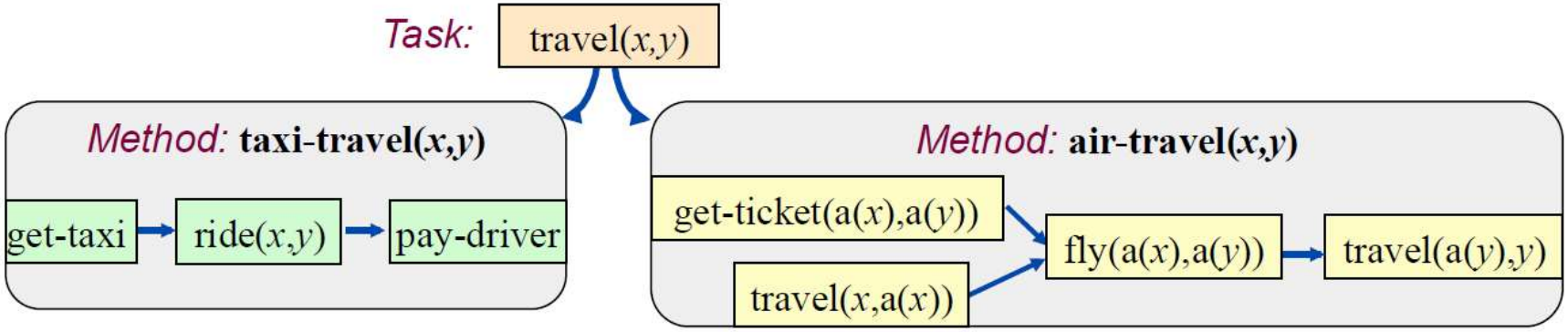
Hierarchical task networks

- Objective: to perform a set of tasks not a set of goals
- Adds notions of tasks, methods, task networks
- Tasks decompose into subtasks thanks to methods



Hierarchical task networks

- Example



Hierarchical task networks

- Method description
  - Non primitive (decomposition into subtasks)
    - » Identifier and parameters
    - » Task performed by the method (with parameters)
    - » A precondition
    - » A sequence of subtasks
- Ex:
  - Method: airtravel(x,y)
  - Task: travel(x,y)
  - Precondition: longdistance(x,y)
  - Subtasks: buy-ticket(a(x), a(y)), travel(x,a(x)), fly(a(x), a(y)), travel(a(y),y)

Hierarchical task networks

- Method description
  - Primitive
    - » Identifier and parameters
    - » Task realized by the method (with parameters)
    - » Operator (with parameters)

- Relationship between non-primitive tasks and methods
  - Decomposed by applying a method
- Relationship between primitive tasks and operators
  - Primitive task is achieved by applying an operator

Hierarchical task networks

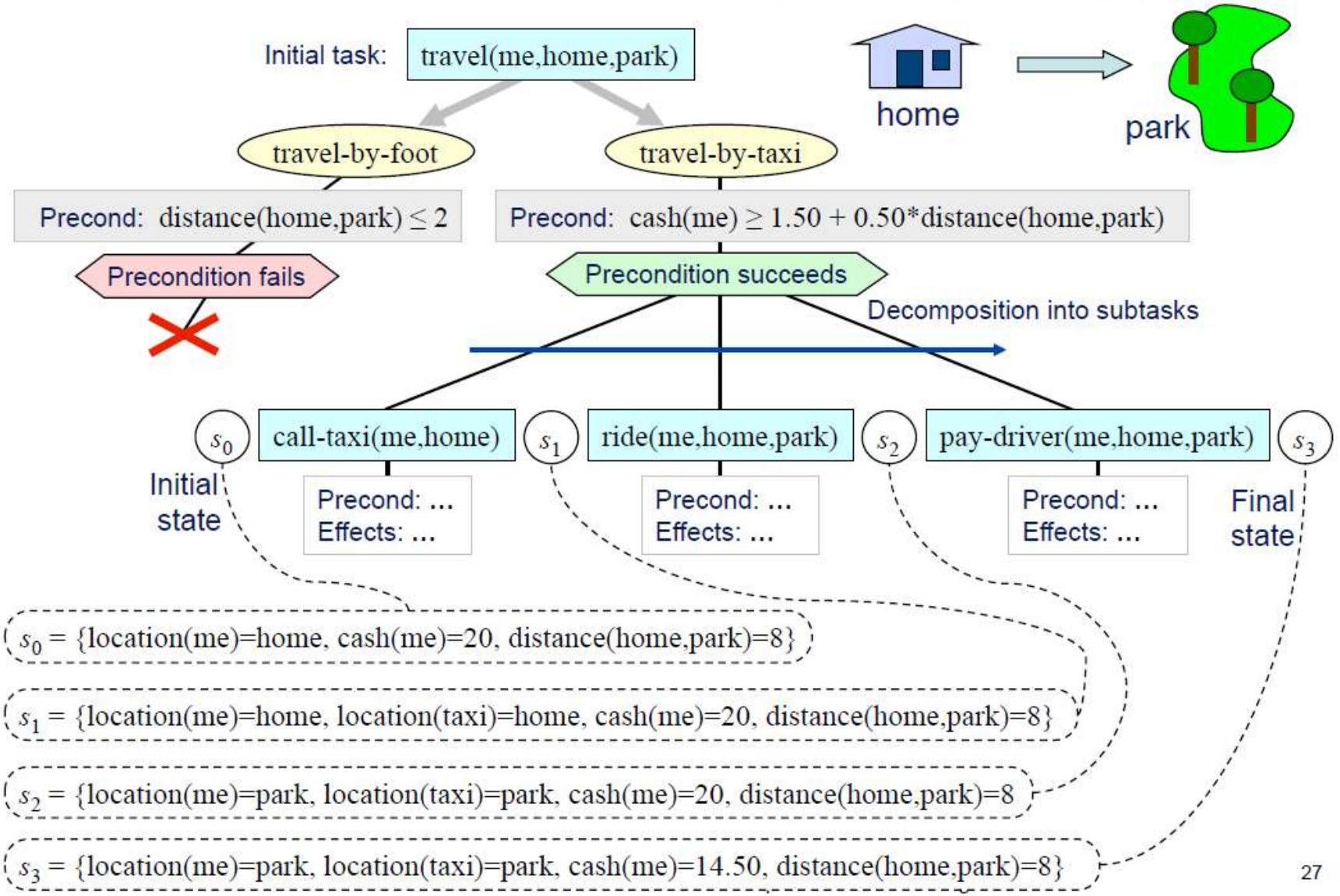
- Domain consists of
  - methods and operators (SHOP- axioms)
- Problem consists of
  - domain, initial state, initial task network (tasks to accomplish, with some ordering of the tasks defined)
- Solution
  - A plan: Totally ordered collection of primitive tasks

Hierarchical task networks

- Planning technique overview
  - Given a list of tasks
  - Use a left first exploration of the decomposition tree
  - When a free variable appears (often the case)
    - » Try multiple instantiations of this variable with objects of the world
  - When a precondition is not satisfied backtrack either by
    - » Trying another free variable instantiation if possible
    - » Returning a failure and trying another decomposition
  - When an operator appear an its precondition is true
    - » Apply the operator to change the state of the world
    - » Continue decomposition tree exploration
- At the end, either a valid decomposition has been found (a plan has been found) or the planner failed.

Planning Problem:

I am at home, I have $20,
I want to go to a park 8 miles away



Hierarchical task networks

- Some implementations exists and can be downloaded
  - SHOP, JSHOP, UMCP
- Mixes long term planning and designer control by method description
- More expressive than classical planning
  - Methods, tasks instead of goals
  - Numerical values in preconditions and effects (ressoutces)
- But not reactive…