Lecture Notes for

Managing and Mining Multiplayer Online Games

Summer Term 2018

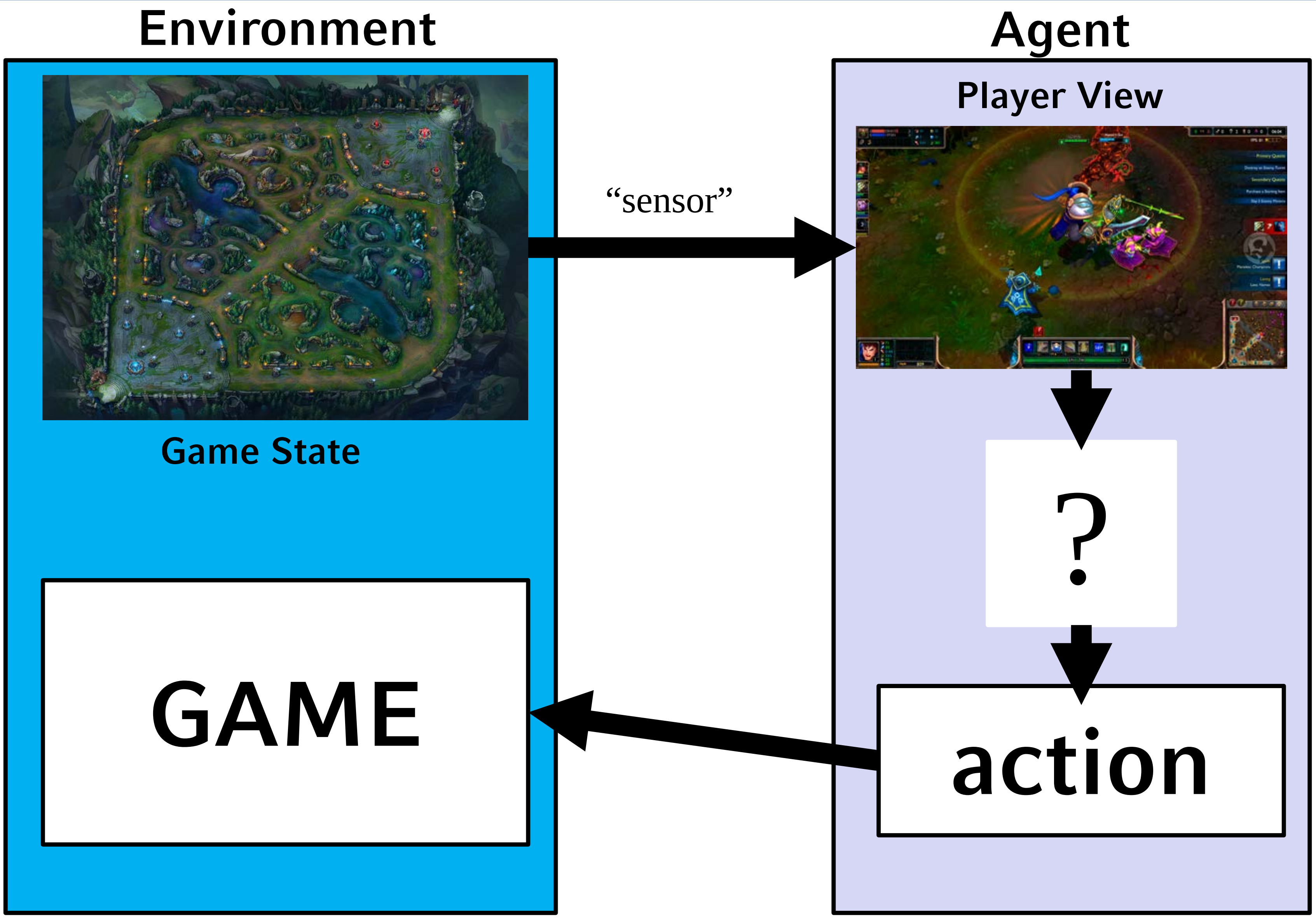
Chapter 9: Artificial Intelligence

http://www.dbs.ifi.lmu.de/cms/VO_Managing_Massive_Multiplayer_Online_Games

Chapter Overview

- What is Artificial Intelligence?
- Environments, Agents, Actions Rewards
- Sequential Decision Making
    - Classical Search
    - Planning with Uncertainty
    - Model-free Reinforcement Learning
    - Monte-Carlo and Temporal Difference Learning
    - Q-Learning
- Adversarial Search
    - Minimax
    - Alpha-Beta Pruning
    - Monte Carlo Tree Search

What is Artificial Intelligence?



Environment

Represents the world in which the agent is acting.
(e.g. a game, a simulation or a robot)

- provides information about the state (e.g. view of Game State)
- receives action and reacts to the them

Properties of Environments

- partially / fully observable
- with known model/ model free
- deterministic / non-deterministic
- single vs. multi-agent
- competitive vs. collaborative
- static / dynamic / semi-dynamic
- discrete / continuous (states and/or actions)

Agents

Autonomous entity within the environment.

types of agents:

- simple reflex agent
  - condition-action-rule
    (example: If car-in-front-is-braking then initiate-braking.)
- model-based reflex agents (add internal state from history)
- goal-based agents (works towards a goal)
- **utility-based agents** (optimizes rewards/minimizes costs)
- **learning agents** (learns how to optimize rewards/costs)

Example: „Autocamp 2000“ (simple reflex agent)

Example for a Bots: (http://www.gamespy.com/articles/489/489833p1.html)

1) If invited by any group => join group
2) If in a group => follow behind the leader
3) If sees a monster => attack
4) If someone says something ending in a question mark => respond by saying "Dude?"
5) If someone says something ending in an exclamation point => respond by saying "Dude!"
6) If someone says something ending with a period => respond by randomly saying one of three things: "Okie“, "Sure“, or "Right on"
7) EXCEPTION: If someone says something directly to you by mentioning your name, respond by saying "Lag."

Example

KillSwitch: [Shouting] Does anyone want to join our hunting party?

Farglik: [Powered by the Autocamp 2000] Dude?

[KillSwitch invites Farglik to join the group.]

[Farglik joins the group]

KillSwitch: We're gonna go hunt wrixes.

Farglik: Right on.

[The group of players runs out, Farglik following close behind. Farglik shoots at every little monster they pass.]

KillSwitch: Why are you attacking the durneys?

Farglik: Dude?

KillSwitch: The durneys, the little bunny things -- why do you keep shooting at them?

Farglik: Dude?

KillSwitch: Knock it off guys, I see some wrixes up ahead. Let's do this.

Farglik: Right on.

[The group encounters a bunch of dangerous wrixes, but they gang up and shoot every one of them.]

KillSwitch: We rock!

Farglik: Dude!

Troobacca: We so OWNED them!

Farglik: Dude!

Example

KillSwitch: Uh oh, hang on. Up ahead are some Sharnaff bulls. We can't handle them, so don't shoot.

Farglik: Okie.

[Farglik shoots one of the Sharnaff bulls.]

[The bull attacks; Trobacca and several other party members are killed before they beat it.]

KillSwitch: You IDIOT! Farglik why did you shoot at them?

Farglik: Lag.

KillSwitch: Well don't do it again.

Farglik: Sure.

[Farglik shoots at another Sharnaff bull.]

[The entire party is slaughtered except for Farglik.]

[... Farglik stands there, alone, for several hours ...]

Planet Fargo- The Automated Online Role-Player
By Dave Kosak
http://www.gamespy.com/articles/489/489833p1.html

Sequential Decision Making

- behavior is a sequence of actions
- sometimes immediate rewards must be sacrificed to acquire rewards in the future

  example: spend gold to build a gold mine
- short-rewards might be very unlikely in a situation
  example: score a goal from your own half in football

=> intelligent behavior needs to plan ahead

Deterministic Sequential Planning

- Set of states $\boldsymbol{S} = \{s_1, .., s_n\}$
- Set of actions ***A(s)*** for each state $s \in S$
- Reward function **R: R(s)** (if negative = cost function)
- Transition function **T: S×A => S: t(s,a) = s'**

  (deterministic case !!)
- this implies:
  - episode = $s_1, a_1, r_1, s_2, a_2, r_2, s_3, a_3, r_3, s_3 \ldots, s_l, a_l, r_l, s_{l+1}$
  - *reward of the episode:* $\sum_{i=1}^{l} \gamma^i r_i$ *with* $0 < \gamma \leq 1$

    (γ=1: all rewards count the same, γ <1: early rewards count more)
  - ***sometimes****: process terminates when reaching a terminal state* s^T *(Game Over !!!) or process end after k moves.*

Deterministic Sequential Planning

- Static, discrete, deterministic, known and fully observable environments:
  - S, A are discrete sets and known
  - t(s,a) has a deterministic state s'
  - Agent knows the current state

- **goal**: find a sequence of actions (path) that maximize the cumulated future rewards.

examples:

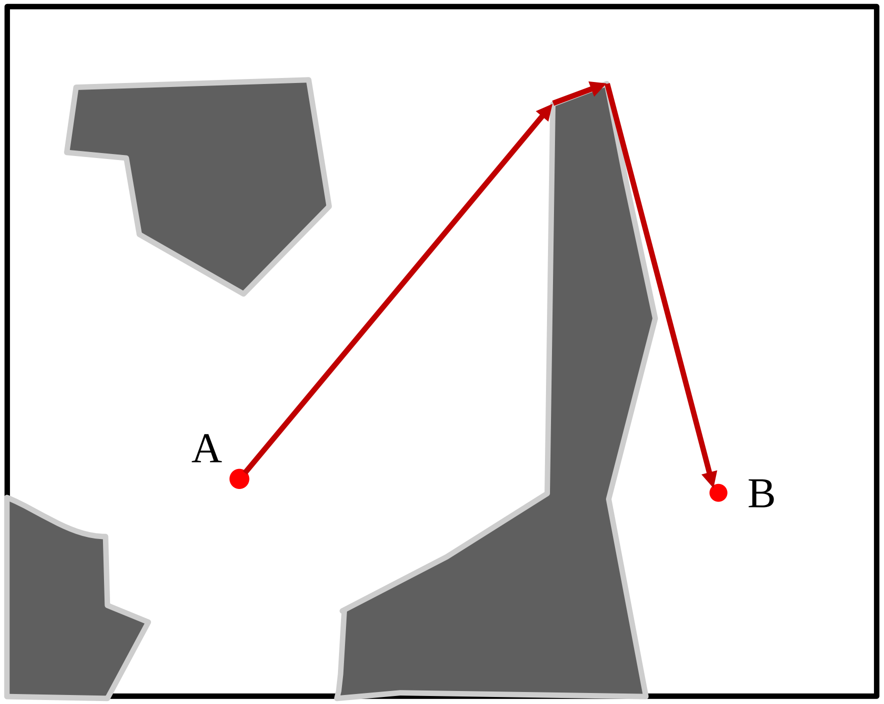
- routing from A to B on the map or find an exit
- riddles like the goat, wolf, cabbage transport problem

Routing in Open Environments

- open environment: 2D Space (IR^2)
- agents can move freely
- obstacles block direct connections
- presenting obstacles with:
    - ***polygons***
    - pixel-presentation
    - any geometric form (Circle, Ellipse, …)

solution for polygon presentation:

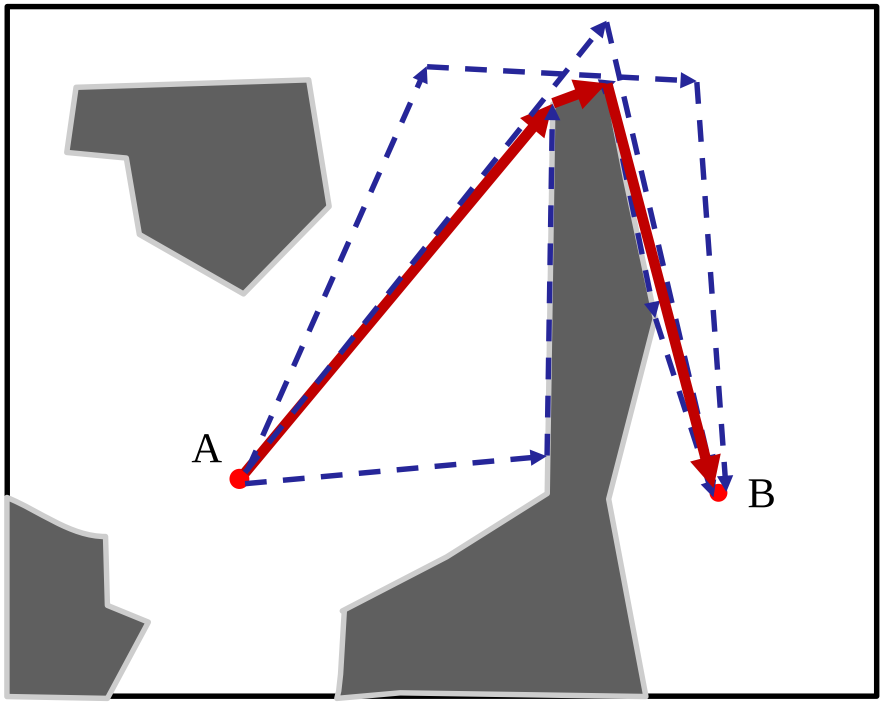
- deriving a graph for the map containing the shortest routes (visibility graph)
- integrating start and goal
- use of pathfinding algorithms like Dijkstra or A*



Visibility Graph

- finding the shortest path in an open environment is a search over an infinite search area
- **solution**: restrict the search area with the following properties of optimal paths:
    - **waypoints of every shortest path are either start, goal or corners of obstacle-polygons.**
    - **paths cannot intersect polygons.**

- The shortest path in the open environment U is also part of the visibility graph $G_U(V,E)$.



Visibility Graph

Environment: *U*

- Set of polygons $U=(P_1, \ldots, P_n)$ **(Obstacles)**
- Polygon P: planar cyclic graph: $P = (V_P, E_P)$

Visibility graph: $G_U(V,E)$

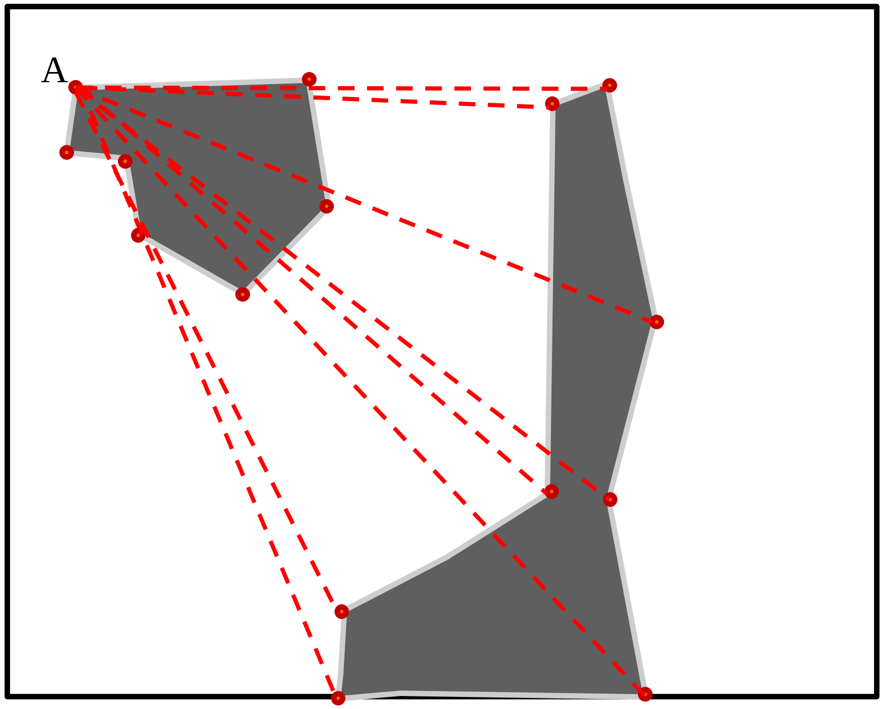
- *Nodes:* Corners of polygons P = $\{V_1, \ldots, V_l)$ in *U:* $V_U = \bigcup_{P \in U} V_P$
- *Edges:* All edges of polygons with all edges of nodes from different polygons that do not intersect another polygon-edge.

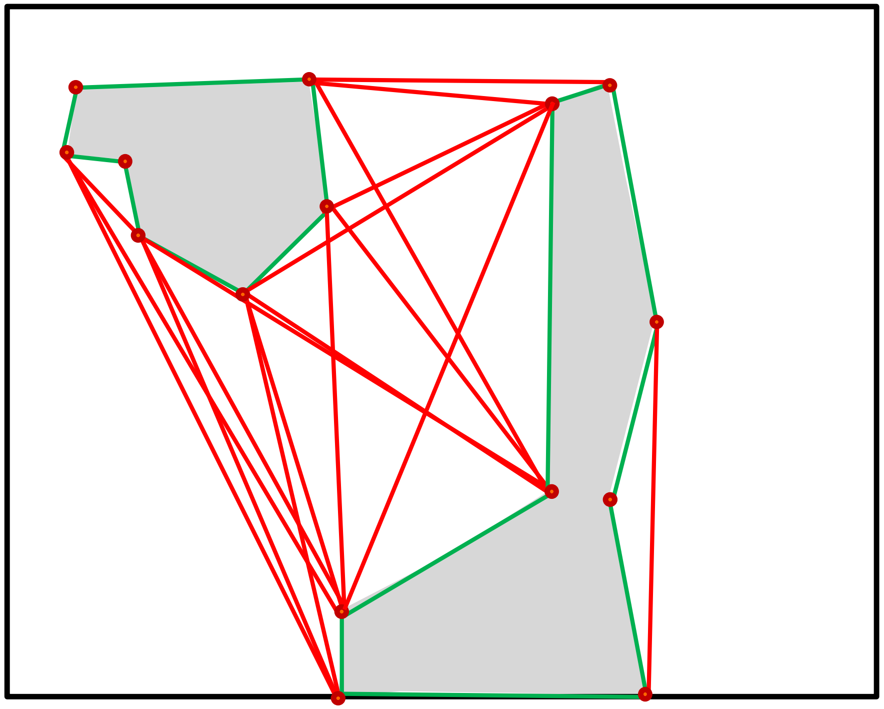
$$E_U = \bigcup_{P \in U} E_P \cup \{(x,y) | x \in P_i \wedge y \in P_j \wedge i \neq j \wedge \underset{P \in U}{\forall} \; \underset{e \in E_P}{\forall} : (x,y) \cap e = \{\}\}$$

Remarks:

- definition applies only to convex obstacles:
- for concave polygons: compute the convex hull of each obstacle and the additional edges
- The definition implies a naive $O(n^3)$ algorithm to construct a visibility graph. Computing a visibility graph can be optimized to $O(n^2)$ (O'Rourke 87)

Example: Visibility Graph



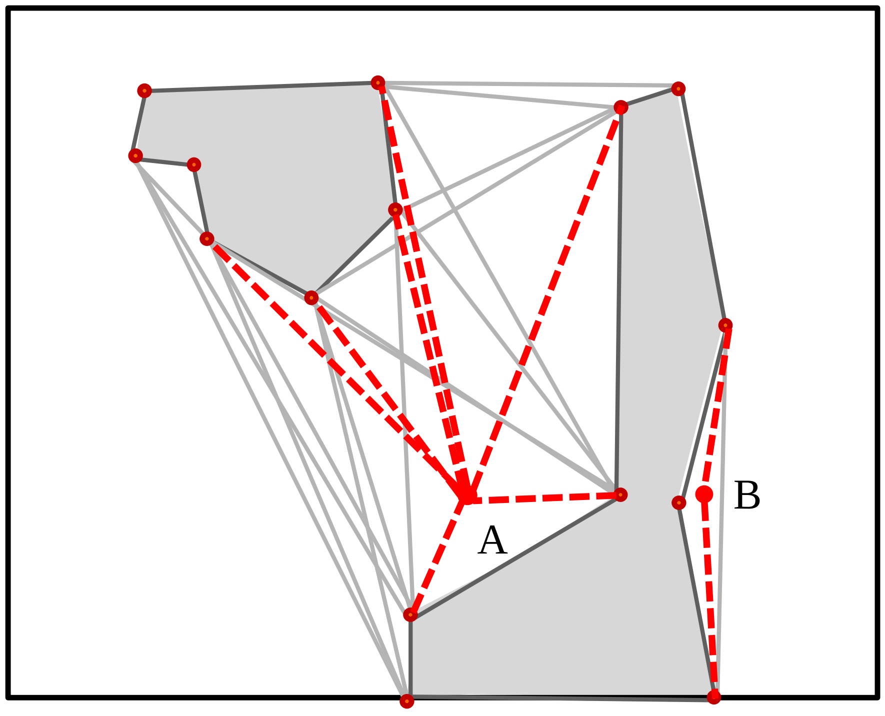


Edges for the node **A** being tested and discarded.

Visibility Graph: Red segments run between polygons. Green segments mark the polygons' borders.

Expansion with Start- and Goal-Nodes

- Visibility graph can be pre-calculated for static environments
- Mobile Objects must be integrated into the graph before calculation
- Inserting start S and goal Z as Point-Polygons
- Connecting the new nodes to with all edges unless an intersection with polygons occurs



Dijkstra's Algorithm

Used Data Structures:

- priorityqueue Q (stores found paths sorted by cost in descending order)
- nodetable T (contains cost for the currently best path for all visited nodes)

Pseudo-Code:

```
FUNCTION Path shortestPath(Node start, Node target)
    Q.insert(new Path(start,0))
    WHILE(Q.notIsEmpty())
        Path aktPath = Q.getFirst()
        IF aktPath.last()  == target THEN                         //target found
            return aktPath
        ELSE
            FOR Node n  in aktPath.last().successor() DO  //extend current path
                Path newPath = aktPath.extend(n)
                IF newPath.cost()<T.get(newPath.last()) THEN        //update optimal path
                    T.update(newPath.last,newPath.cost)
                    Q.insert(newPath,newPath.cost)
                ENDIF
            ENDDO
        ENDIF
    ENDWHILE
    RETURN NULL                                               //start and target not connected
ENDFUNCTION
```

A*-Search

- Dijkstra's algorithm uses no information about the direction to the target

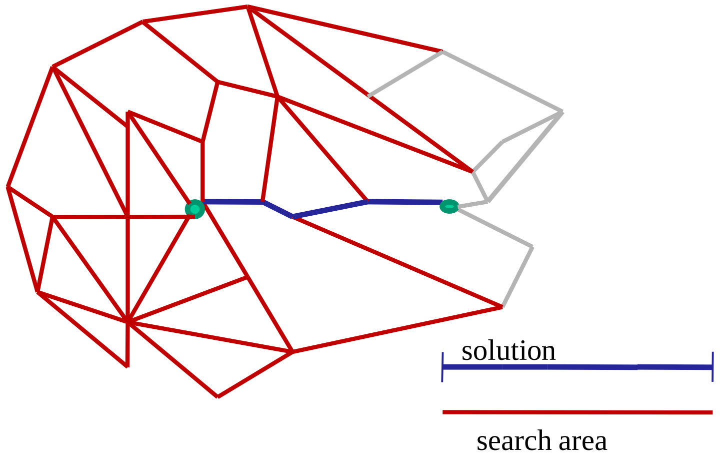
  => the search expands into all directions until the target is found
- A*-Search formalizes the direction into an optimistic forward approximation:

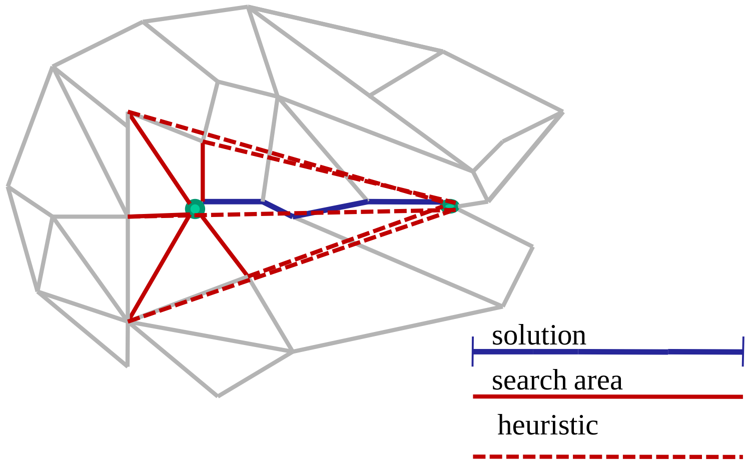
  h(n, target) for each node ***n***
- ***h(n,target)*** indicates a lower bound for the minimum cost to reach the target
- improve the search order by sorting by minimal total cost to reach the target
- allows to prune path ***P*** if:

  P.cost()+h(pfad1.last(),target) > bestPath.cost()

- basic heuristic for network distance:

  Euclidian distance between current position and target position.
  (a straight line is always the shortest connection)





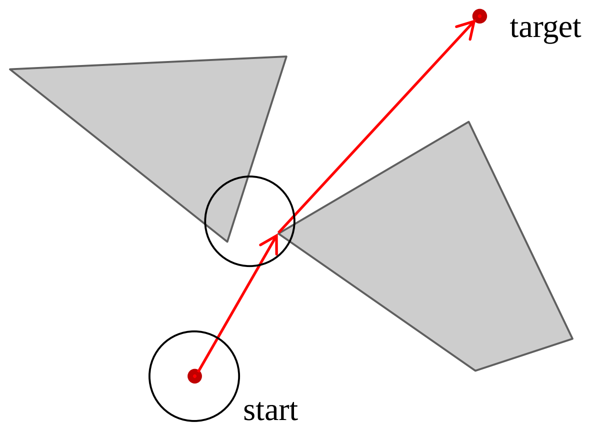
Pseudo-Code: A*-Search

Peseudo-Code: A*-Search

```
FUNCTION Path shortestPath(Node start, Node target)
  Q.insert(new Path(start),0)
  WHILE(Q.notIsEmpty())
        Path aktPath = Q.getFirst()
        IF aktPath.last()  == target THEN          //found result
            return aktPath
        ELSE
            FOR Node n  in aktPath.last().successor() DO    //expanding the current path
                Path newPath = aktPath.extend(n)
                IF newPath.cost()<T.get(newPath.last()) THEN //update if optimal so far
                    T.update(newPath.last, newPath.cost())
                    Q.insert(newPath, newPath.cost() +h(newPath.getLast(), target))
                ENDIF
            ENDDO
        ENDIF
  ENDWHILE
  RETURN NULL            //there is no path
ENDFUNCTION
```

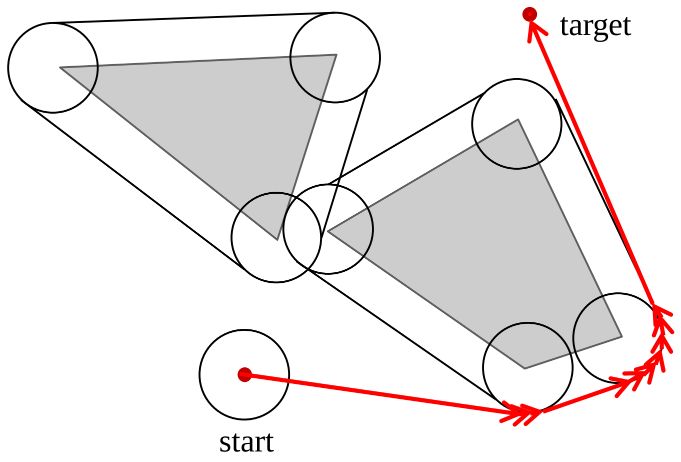
Visibility Graph for extended objects

- agents usually have a spatial expansion: Circle or Polygon
- visibility graph is only feasible for point objects
- adjust the visibility graph: expand obstacle polygons by the spatial expansion of the agent (Minkowski Sum)



problem with this solution:

- for circular expansion: circles have an infinite number of edges
  => visibility graph is not derivable
- For Polygon-Environment: object rotation should be considered
  => Every rotation requires a separate extension



Visibility Graph for extended Objects

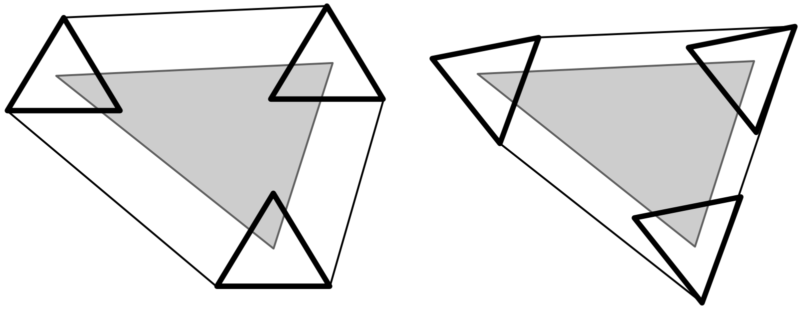
Solution Approach:

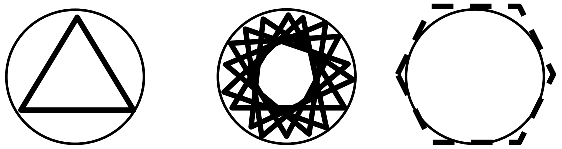
- polygons are approximated by the body of rotation => Circle
- circles are approximated by minimal surrounding polygons (MUP)

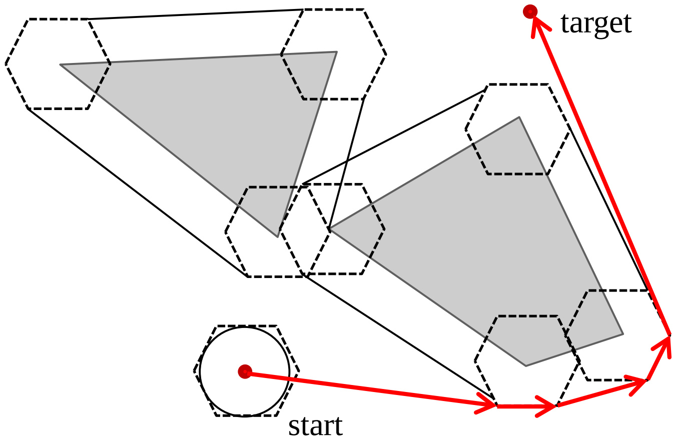
  => e.g. hexagon, octagon
- form Minkowski sum with the MUPs and derive visibility graph.

Remarks:

- paths are not optimal
- passages are considered conservative
- curves are taken angular
- MMO should only have a limited selection of agent extensions because each requires it's own graph


Minkowski sum of varying rotations of the same shape

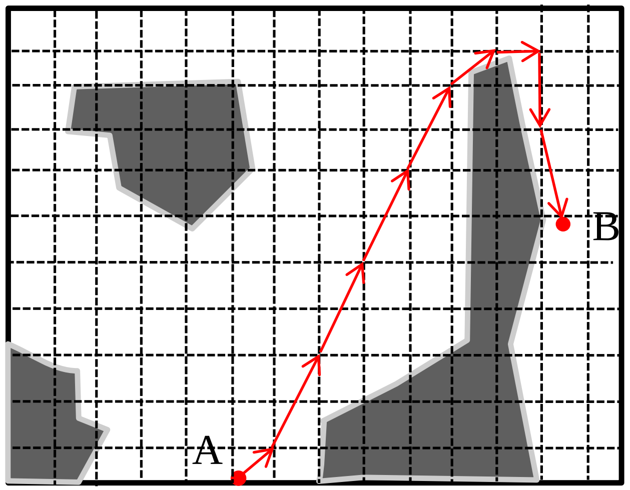

double approximation by building the body of rotation and the minimal surrounding Hexagon

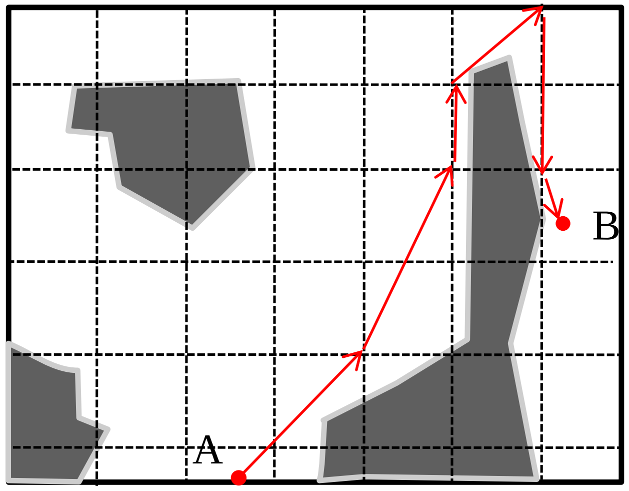


More Pathfinding Methods

Other Methods:

- approximate polygons with polygons having less corners
- hierarchic routing for longer routes
- precalculate and store shortest paths
- use a grid based graph, overlay the map with a grid and route over cell centers => decent approximation
- use sampled graphs in open environments (cmp. task-motion planning in robotics)





Markov Decision Process

Problem: We might not know the outcome of an executed action

- other players might act unpredictable
- game integrate random processes

$\Rightarrow$ **t(s,a) is stochastic** T: $P(s'|s,a)$ for all $s,s' \in S$, $a \in A$

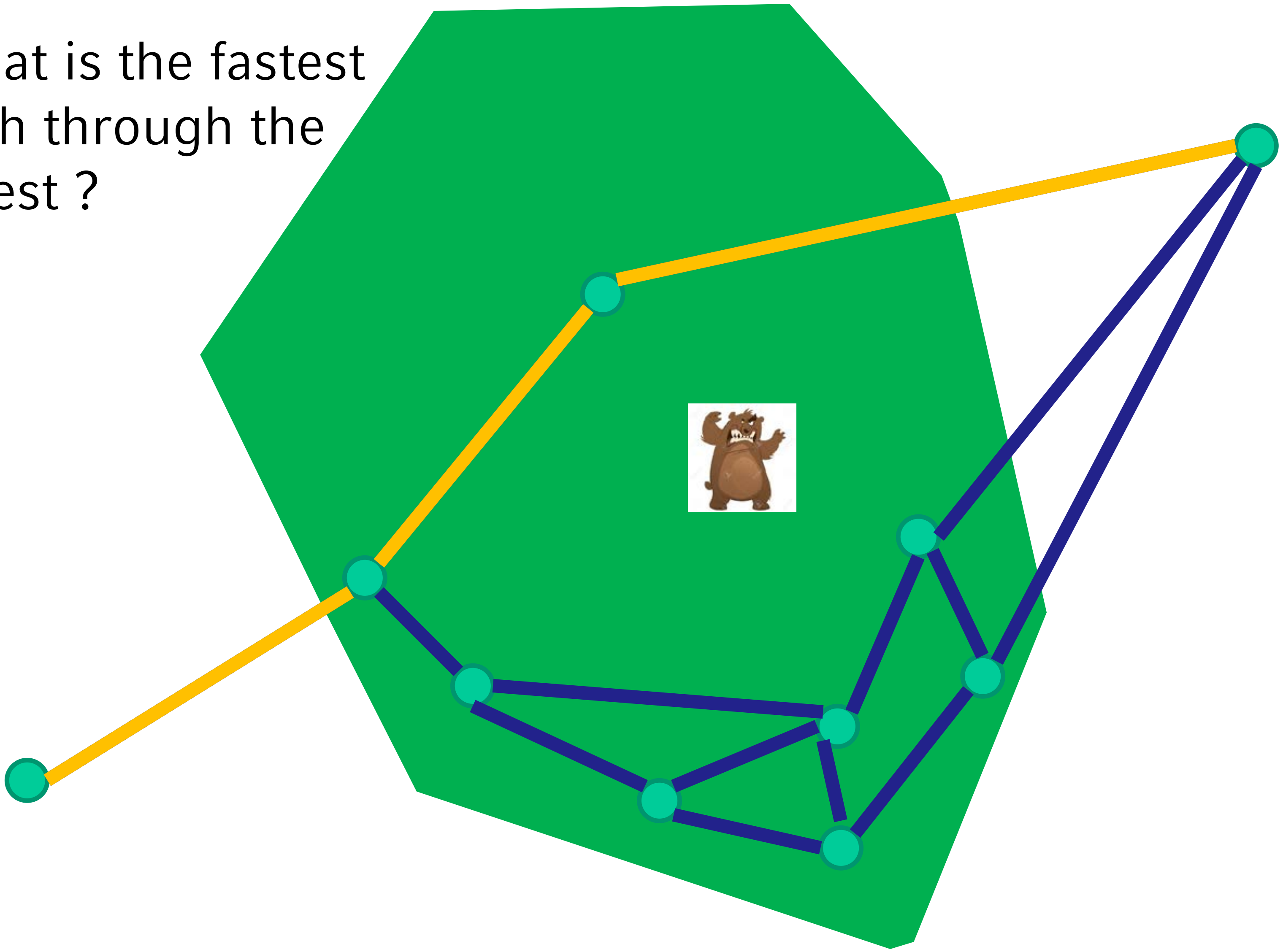
implications:

- we do not know what the future state after performing action a is.
- the reward R(s) gets stochastic as well
- when searching a terminal state, there is no secure path leading us to the target

=> We need an action for each situation we might encounter and not just the one on the path

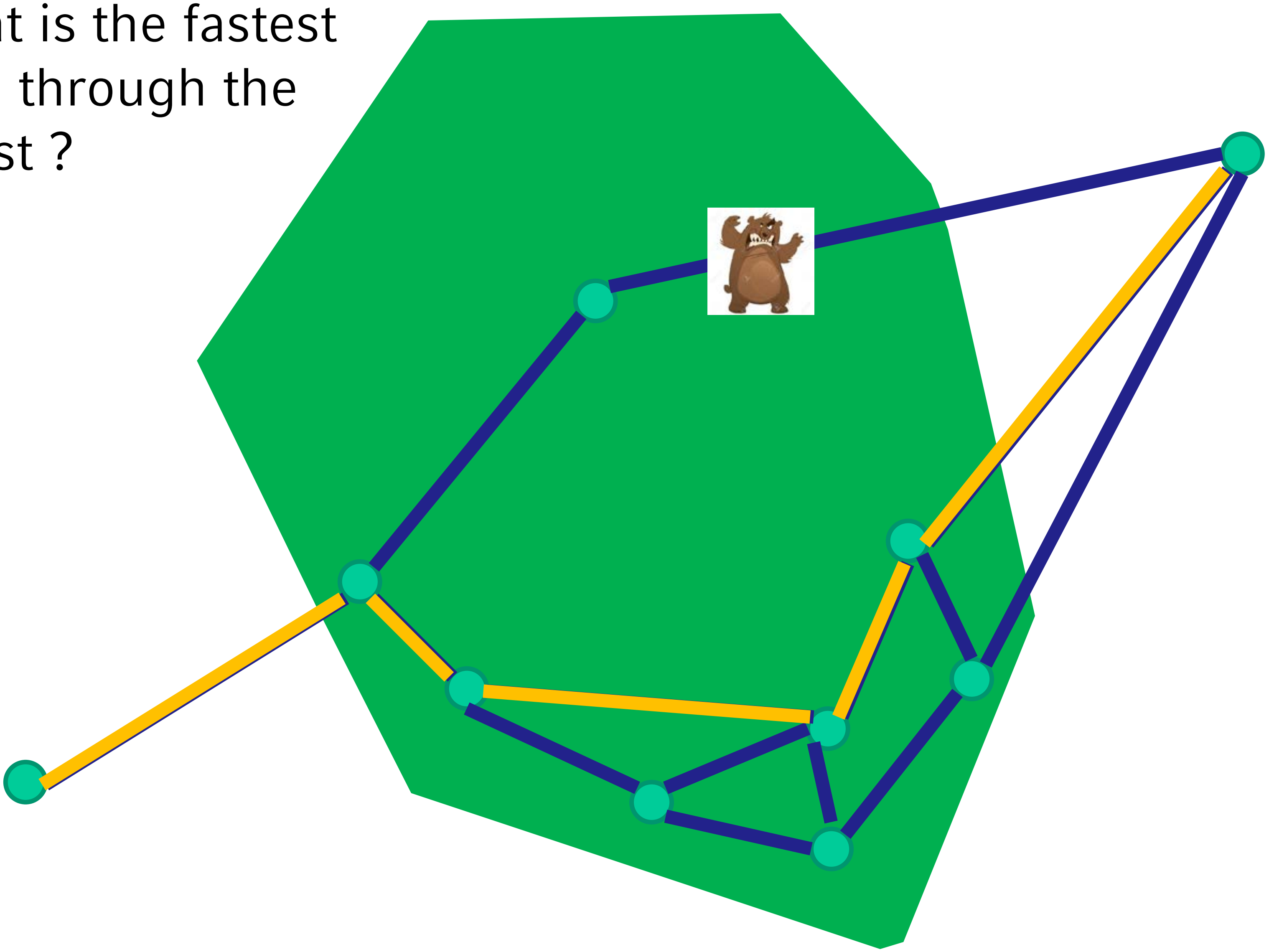
Motivation non-deterministic Routing

What is the fastest path through the forest ?



Motivation non-deterministic Routing

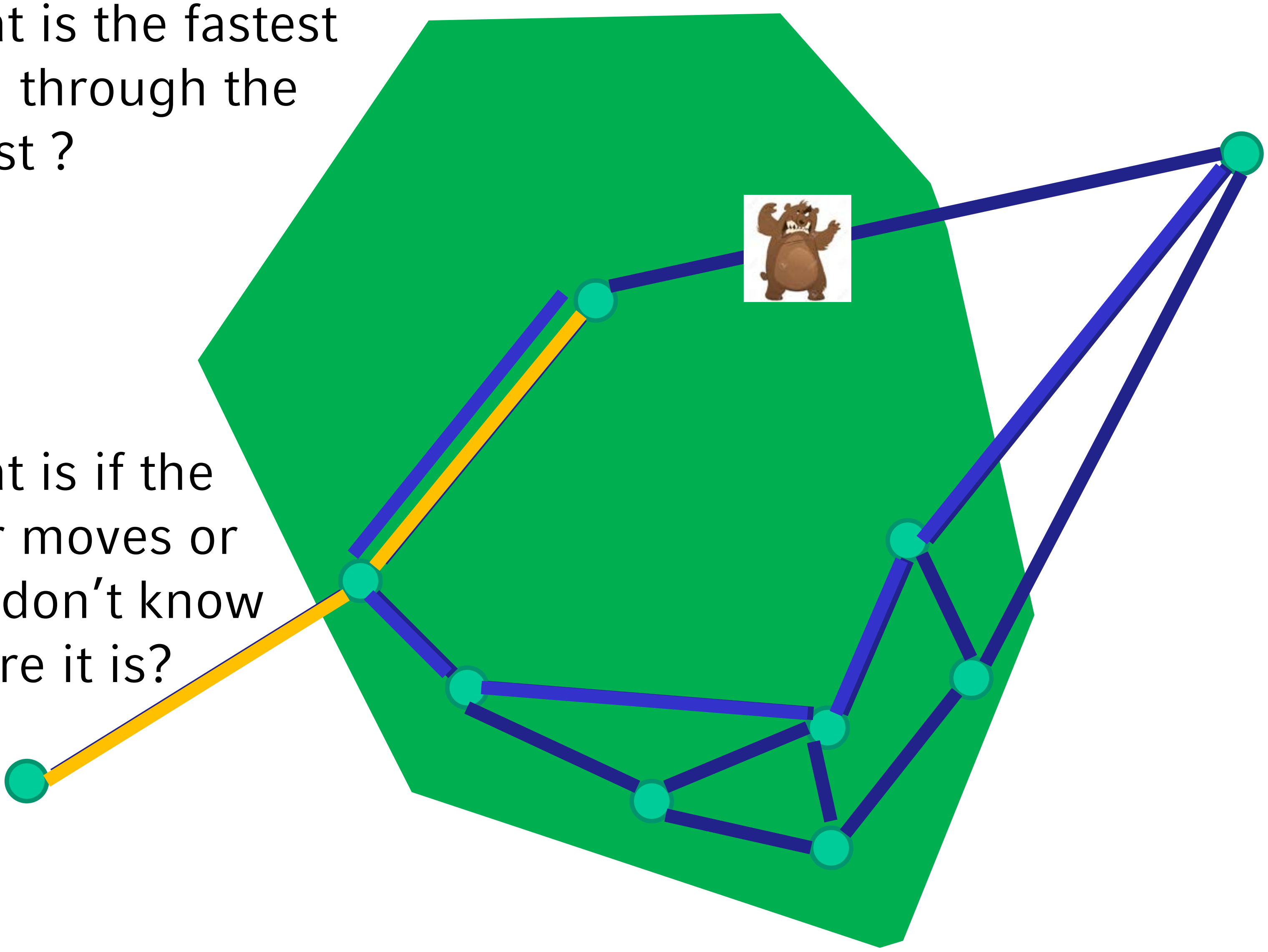
What is the fastest path through the forest ?



Motivation non-deterministic Routing

What is the fastest path through the forest ?

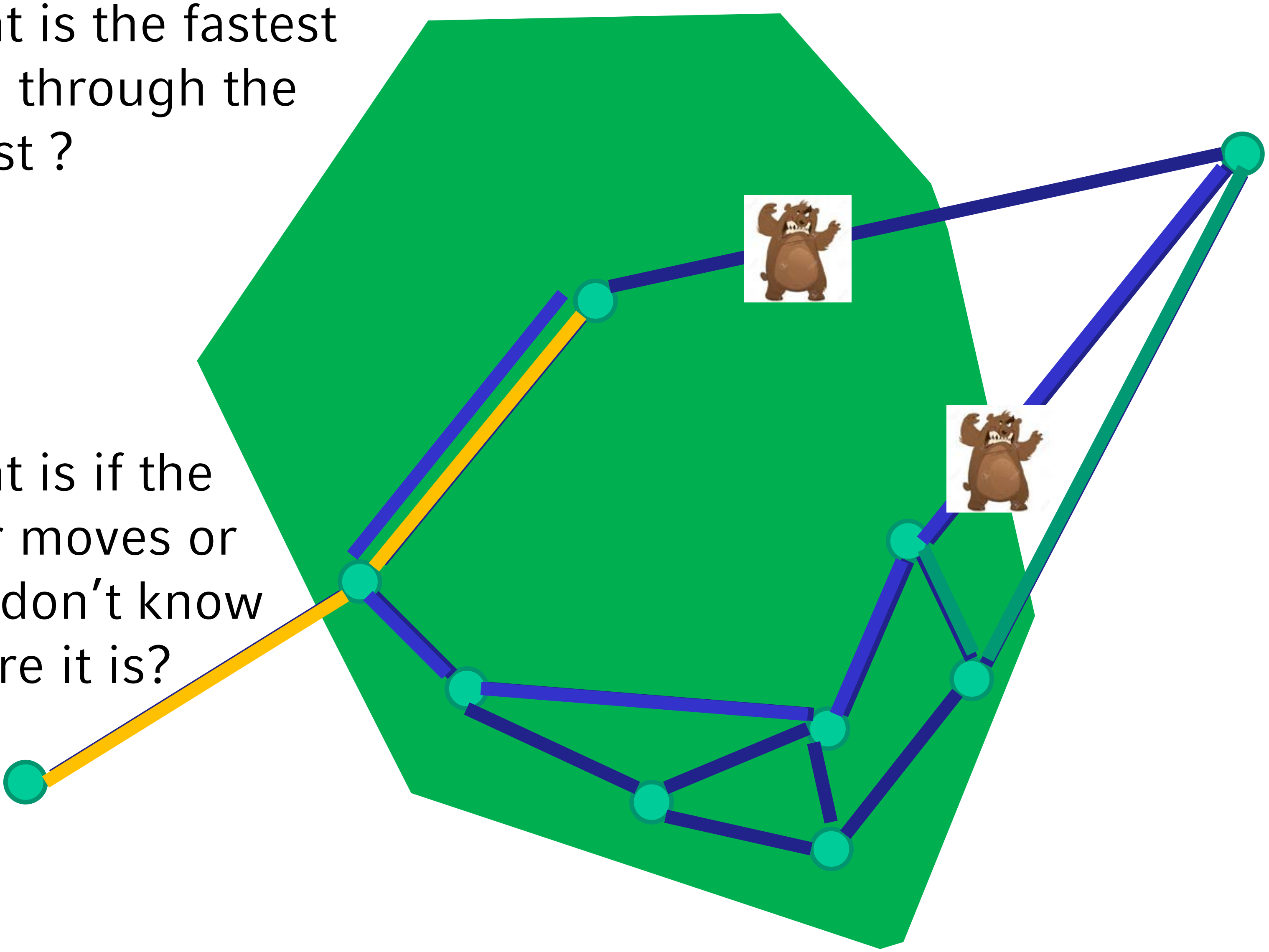
What is if the bear moves or you don’t know where it is?



Motivation non-deterministic Routing

What is the fastest path through the forest ?

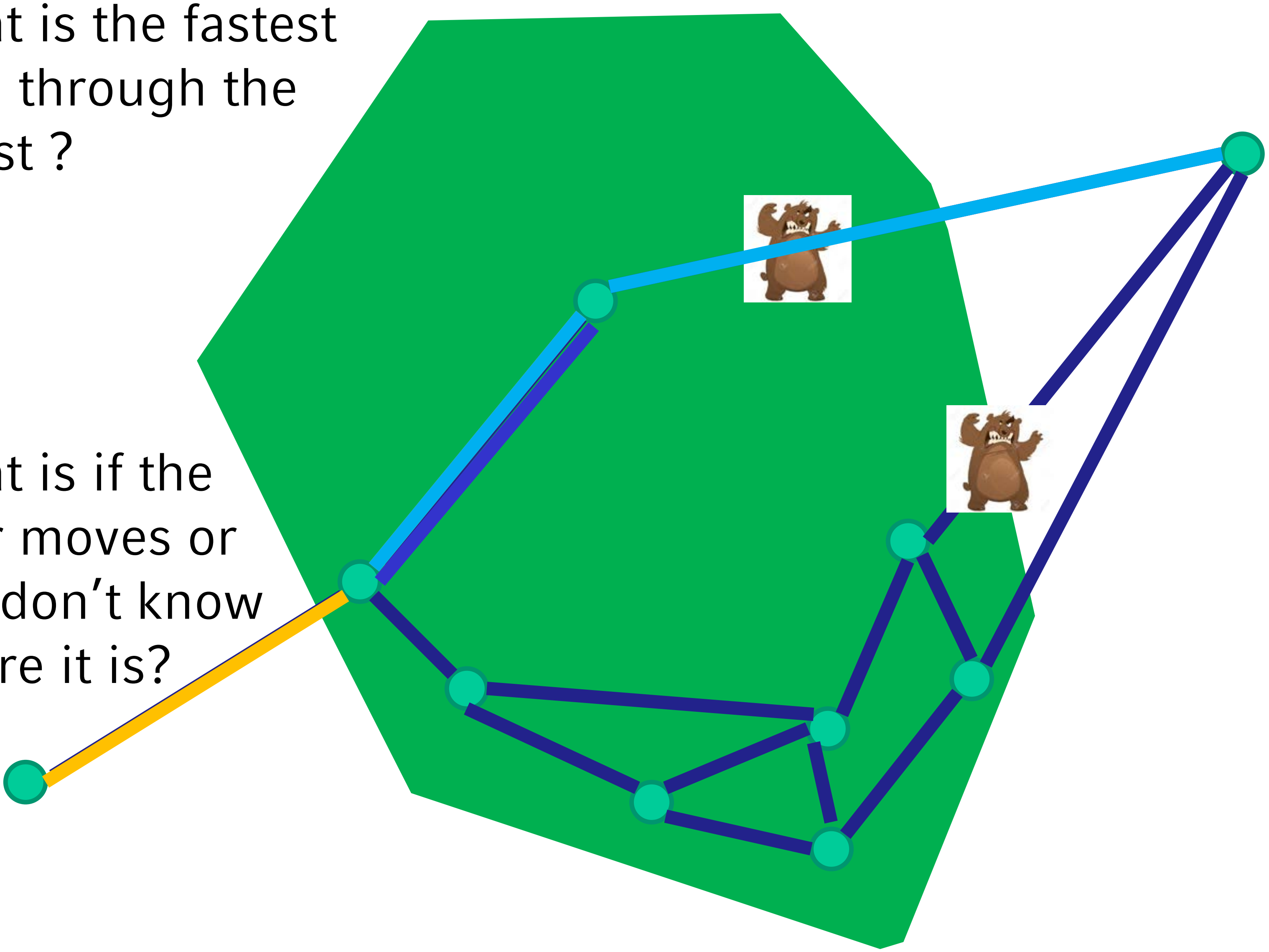
What is if the bear moves or you don’t know where it is?



Motivation non-deterministic Routing

What is the fastest path through the forest ?

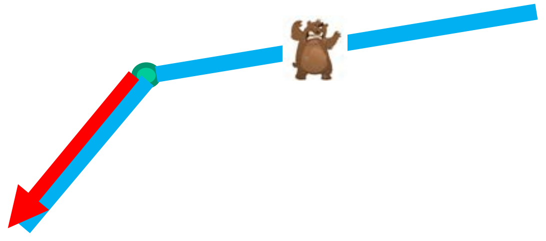
What is if the bear moves or you don't know where it is?

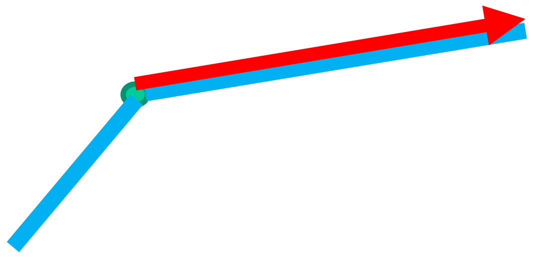


Policies and Utilities

- in **dynamic**, discrete, **non-deterministic**, known and fully observable environments
- a policy π is a mapping defining for every state $s \in S$ an action $\pi(s) \in A(s)$ (agent knows what to do in any situation)
- Stochastic Policies: Sometimes it is beneficial to vary the action then $\pi(s)$ is a distribution function over A(s). (think of a game where strictly following a strategie makes you predictable)

Example:


state: bear is there


state: bear is absent

Bellman's Equations

- What is the reward of following π?

  Utility $U^{\pi}(s)$: expected reward when following π in state s

  - $U^{\pi}(s) = E[\sum_{t=0}^{\infty} \gamma^t R(s_t) \,|s_0 = s, \pi]$
  - $U^{\pi}(s) = \gamma^0 R(s) + E[\sum_{t=1}^{\infty} \gamma^t R(s_t) \,|s_0 = s, \pi]$
  - $U^{\pi}(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s, \pi) U^{\pi}(s')$

    (Bellmann Equation)

- What is the optimal policy π*?

  Bellman Optimality Equation:

$$U^{\pi*}(s) = R(s) + \gamma \, arg\max_{a \in A} \sum_{s' \in S} P(s'|s, a) \, U^{\pi*}(s')$$

Finding optimal Policies: Policy Iteration

- We are looking for the optimal policy.
- The exact utility values are not relevant if one action is clearly the optimal.

- **Idea**: Alternate between:
  - **Policy evaluation**: given a policy, calculate the corresponding utility values
  - **Policy improvement**: Calculate the policy given the utility values $\pi(s) = \underset{a \in A}{arg\max} \sum_{s' \in S} P(s'|s,a)\, U^*(s')$

Policy Evaluation

- Policy evaluation is much simpler than solving the bellman equation

$$U^{\pi}(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s,\pi)\, U^{\pi}(s')$$

- Note that the non-linear function “max” is not present
- We can solve this by standard algorithms for system of linear equations
- For large state spaces solving systems of linear equations takes a long time ($O(n^3)$)
- In large state spaces a simplified Bellman update for k times can be more performant

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s'|s,\pi_i)\, U_i(s')$$

Policy Iteration

```
repeat
    $U \leftarrow \mathrm{PolicyEvaluation}(\pi, U, mdp)$
    $unchanged? \leftarrow true$
    for each state s in S do
        if $\max_{\boldsymbol{a} \in \boldsymbol{A}} \sum_{\boldsymbol{s'}} \boldsymbol{P}(\boldsymbol{s'}|\boldsymbol{s},\boldsymbol{a})\, \boldsymbol{U}[\boldsymbol{s'}] > \sum_{\boldsymbol{s'}} \boldsymbol{P}(\boldsymbol{s'}|\boldsymbol{s},\boldsymbol{\pi})\, \boldsymbol{U}[\boldsymbol{s'}]$ then
            $\boldsymbol{\pi}[\boldsymbol{s}] \leftarrow \operatorname{argmax}_{a \in A} \sum_{\boldsymbol{s'}} \boldsymbol{P}(\boldsymbol{s'}|\boldsymbol{s},\boldsymbol{a})\, \boldsymbol{U}[\boldsymbol{s'}]$
            $\boldsymbol{unchanged}? \leftarrow \boldsymbol{false}$
until $unchanged?$
return $\pi$
```

Value Iteration

- if we use Bellman updates anyway, we can join both steps
- update Bellman optimality equation directly:

$$U^*(s) = R(s) + \gamma \, arg\max_{a \in A} \sum_{s' \in S} P(s'|s,a)\, U^*(s')$$

- Non-linear system of equations
- Use Dynamic Programming
    - Compute utility values for each state by using the current utility estimate
    - Repeat until it converges to U^*
    - convergence can be shown by contraction

Value Iteration

```
repeat
        $U \leftarrow U'$
        $\delta \leftarrow 0$
        for each state s in S do
                $U'[s] \leftarrow R(s) + \gamma \max_{a \in A} \sum_{s'} P(s'|s,a)\, U[s']$
                if $|U'[s] - U[s]| > \delta$ then $\delta \leftarrow |U'[s] - U[s]|$
until $\delta < \frac{\epsilon(1-\gamma)}{\gamma}$
return $U$
```

MDP Synopsis

- MDPs rely on a Markov model with assumptions about: states, actions, rewards, transition probabilities, etc.
- if all the information is available, computing the optimal policy does not require any learning samples
- Transitions probabilities are usually not defined but have to be estimated based on observations
- usually observations $\neq$ states
  - $\Rightarrow$ partially observable MDP, estimate belief states (compare HMM)
  - $\Rightarrow$ set of possible states and transitions is unknown

Can we learn on observations only without making an model assumptions ?

Model-Free Reinforcement Learning

If we don't have a model, what do we have:

NOTE: s is a state description, but S does not need to be known.

1. Sample Episodes:
   - episode = $s_1, a_1, r_1, s_2, a_2, r_2, s_3, a_3, r_3, s_3 \ldots, s_l, a_l, r_l, s_{l+1}$
   - *reward of the episode:* $\sum_{i=1}^{l} \gamma^i r_i$ *with* $0 < \gamma \leq 1$
   - episode might end with terminal state

2. Queryable Environments:
   - Agent selects an action $a \in A(s)$ and receives on new state s', R(s'), A(s') from the Environment.
   - Allows to generate episodes

Monte-Carlo Policy Evaluation

- for a know policy π and a set of complete sample episode X following π
- let X(s) be the set of (sub-)episodes starting with s
- to estimate utility U^π(s) average over the expected reward:

$$\mathrm{U}(s) = \sum_{x \in X(s)} \frac{\sum_{i=1}^{l} \gamma^i r_i}{\lfloor X(s) \rfloor}$$

- if X(s) gets sufficiently large for all $s \in S$: $U(s) \rightarrow U^\pi(s)$
- if new episodes, compute incremental mean:

$$\mu_k = \frac{1}{k} \sum_{j=1}^{k} x_j = \frac{1}{k}\left(x_k + \sum_{j=1}^{k-1} x_j \right) = \frac{1}{k}(x_k + (k-1)\mu_{k-1})$$
$$= \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})$$

- if environments is non-stationary, limit weight of old episodes:

$$\mathrm{U}(s_t) \leftarrow \mathrm{U}(s_t) + \alpha\big(R(x) - \mathrm{U}(s_t)\big)$$

Temporal Difference Learning

problem: Can we still learn if episodes are incomplete?

- the later part of $\sum_{i=1}^{l} \gamma^i r_i$ is missing
- in the extreme case we just have 1 Step: s_t, a, r, s_{t+1}

=> Temporal Difference Learning

- idea similar to incremental Monte-Carlo learning:

$$\mathrm{U}(s_t) \leftarrow \mathrm{U}(s_t) + \alpha\big(R(x) - \mathrm{U}(s_t)\big)$$

- Policy Evaluation with Temporal Difference (TD) Learning:

$$\mathrm{U}(s_t) \leftarrow \mathrm{U}(s_t) + \alpha\big(R(s_{t+1}) - \gamma \mathrm{U}(s_{t+1}) - \mathrm{U}(s_t)\big)$$

- TD target: $R(s_{t+1}) - \gamma \mathrm{U}(s_{t+1})$
- TD error: $R(s_{t+1}) - \gamma \mathrm{U}(s_{t+1}) - \mathrm{U}(s_t)$
- each step estimates the mean utility incrementally

Policy Optimization

Idea: adapt Policy Iteration
(evaluate policy and update greedily)

- greedy policy update of U(s) requires MDP:

$$\pi'(s) = R(s) + argmax_{a \in A(s)} P(s'|s,a) U(s')$$

- Q-Value Q(s,a): If we choose action a in state s what is the expected reward?
  => We do not need to know where action a will take us!
- Improving Q(s,a) is model free:

$$\pi'(s) = argmax_{a \in A(s)} Q(s,a)$$

- Adapt the idea of Policy Iteration:
  - Start with a default policy
  - evaluate policy (previous slide)
  - update policy: e.g. with greedy strategy

Samples and Policy Updates

Problem: After updating a policy, we need enough samples following the policy.

- real observed episodes usually do not cover enough policies (episodic samples are policy dependend)

$$s_1, a_1, r_1, s_2, a_2, r_2, s_3, a_3, r_3, s_3 \dots, s_l, a_l, r_l, s_{l+1}$$

- we need to dynamically sample from an environment:
  - measure reaction of physical world (e.g. robotics..)
  - build simulations which mimic the physical world
  - in Games: let the agent play and learn !!!
- We need a strategy for sampling these s,a pairs.
- s is often determined by the environment as result of the last action. (The game is in state after the last move.)

Learning on a Queryable Environment

- we can generate as much samples as possible
- environment might be non-deterministic:
  - same state s and action a => different outcomes s' and R(s')
  - multiple samples for the same (s,a) might be necessary

- How to sample over the state-action space?
  - **exploit**: If we find a good action keep it and improve the estimate of Q(s,a). Usually, it's a waste of time to optimize Q(s,a) for bad actions.
  - **explore**: Select unknown or undersampled actions
    - a low Q(s,a) need not mean that the option is bad, maybe it is just underexplored.
    - try out new things might lead to a even better solution

ε-Greedy Exploration

- makes sure that sampling considers new actions
- when sampling:
  - With probability 1-ε choose greedy action
  - with probability ε chose random action
- Sampling policy:

$$\pi(a|s) = \begin{cases} \frac{\varepsilon}{m} + (1-\varepsilon) & if\ a = argmax_{a \in A(s)} Q(s,a) \\ \frac{\varepsilon}{m} & otherwise \end{cases}$$

- achieves that Q-values improve and guarantees that all actions are explored if optimized long enough

On-Policy and Off-Policy Learning

Which Q(s,a) is used for sampling an action?

on-policy learning: Sample with respect to the currently learned policy. ***example***: SARSA

off-policy learning: Sampling is done based on a behavioral policy which is different from the learned policy.

example: Q-Learning

Implication:

- For off-policy learning, exploration is usually just done for the behavioral policy.
- For on-policy methods, exploration must be part of the learned policy.

Q-Learning

- standard off-policy learning method
- Given a behavioral policy π_b, learn the policy π_l by the following learning update:

$$Q(s, \pi_b(s)) \leftarrow Q(s, \pi_b(s)) + \alpha\big(R(s) + \gamma\, Q(s', \pi_l(s')) - Q(s, \pi_b(s))\big)$$

- Usually we want to learn the optimal policy, thus:

$$\pi_l(s) = \underset{a \in A}{\operatorname{argmax}}\, Q(s, a)$$

- For behavioral policy, choose ε-Greedy

Q-Learning Algorithm

```
init Q(s,a)$\forall s \in S, a \in A$
for n episodes:
  init s
  repeat until episode is finished:
    choose a from A(s) with $\pi_b$
    s’,r = query_Env(s,a)
    Q(s,a)$\leftarrow$ Q(s,a)+$\alpha$(r+$\gamma$max$_a$Q(S’,a)-Q(s,a))
    s $\leftarrow$ s’
  until s is terminated
//terminal state or finite horizon is reached
```