Theory and Practice of Artificial Intelligence

Introduction: What is Artificial Intelligence?

Daniel Polani

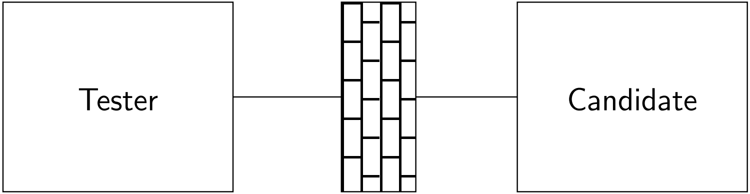
School of Computer Science
University of Hertfordshire

March 9, 2017

Is it AI?

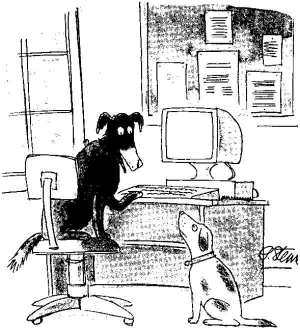
1. text editor
2. searching for a name/address/occupation record in a database
3. chess and go playing programs
4. language translation
5. robot control
6. puzzle solvers
7. Turing test contenders

The Turing Test



- terminal communication with unknown partner
- no way of identifying partner
- **Question:** is partner human or not?
- **See:** e.g. (Saygin et al. 2000)

The Turing Test II



"On the Internet, nobody knows you're a dog."

On the internet, nobody knows you are a dog!

New Yorker Magazine, July 1993

Example: RoboCup

RoboCup: the Robot Soccer World Championship
Simulation League: Humanoid Robots playing soccer

Theory and Practice of Artificial Intelligence

Search

Daniel Polani

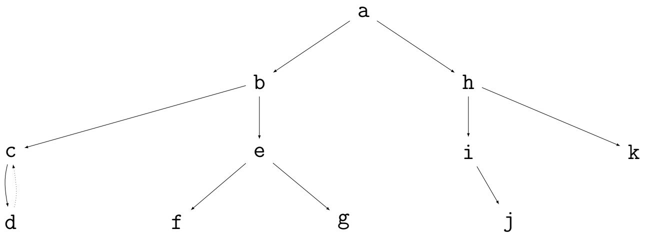
School of Computer Science
University of Hertfordshire

March 9, 2017

Depth-First Search

finds solution path `Sol` from given node `N` to some goal node (there can be several):

- if `N` is a goal node, `Sol=[N]` else
- if there is a successor node `N1` of `N` such that there is a path `Sol1` from `N1` to a goal node `Sol=[N,Sol1]`



Breadth-First Search

finds solution path `Sol` from given node `N` to some goal node:

- if `N` is a goal node, `Sol=[N]` else
- generate one-step extension of paths in candidate lists, adding extensions to that list.
- more detailed: given list of candidate paths
  - if first path contains goal node as head, solution
  - else
    - remove first path from candidate list
    - generate set of one-step extensions
    - append them to list of candidates
    - call breadth-first search on this list

Best-First Search

- breadth-first selects the shortest path
- best-first selects the least “costly” path

For that

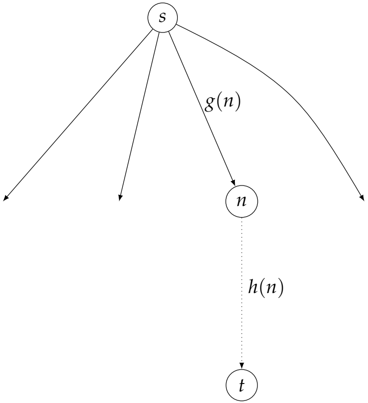
define $c(n, n')$ as cost for moving from node n to n'

Heuristic Search

Heuristics

- consider a heuristic estimator f
- $f(n)$ estimates "cost" of n, i.e. the cost of best solution path from start node s to some goal node, say t, provided that path goes via n

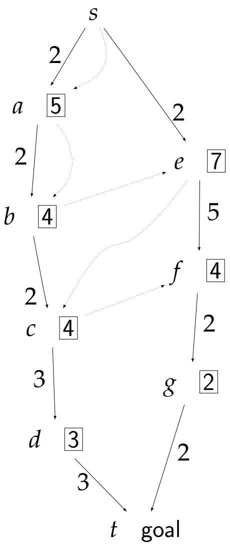
$$f(n) = \underbrace{g(n)}_{\substack{\text{actual cost from } s \text{ to } n \\ \text{not necessarily optimal,} \\ \text{estimate of minimal cost} \\ \text{from } s \text{ to } n}} + \underbrace{h(n)}_{\substack{\text{guesswork!} \\ \text{no universal solution}}}$$



Best-First Idea

Competing Subtrees: among
competitors, only one subtree is active at a time: the most promising, i.e. that with the lowest f-value switch to alternative, if that changes

Example: $f(X) = g(X) + h(X)$
Budget of subtree spent until exhausted, switching to other tree



Best-First Search

Best-First Search

finds cheapest solution path `Sol` from given node `N` to some goal node:

1. if `N` is a goal node, `Sol=[N]` else
2. initialize heap of candidate paths (sorted according to cost, *best first*), containing `[N]`
3. pop head of heap (best candidate solution so far) as candidate solution `C`
4. if `C` has node as head, solution
5. generate all one-step extensions of `C`, add them to heap
6. go to 3

Admissibility

Def.: A search algorithm is *admissible* if it always produces an optimal solution.

Note: Above algorithm immediately produces an optimal solution if for each node n, $h^*(n)$ is the cost of an optimal path from n to some goal node.

Note: The Best-First algorithm is a variant of the noted A^* algorithm.

Theorem: Best-First is admissible if it uses an *optimistic* heuristic h, i.e. h with

$$h(n) \leq h^*(n)$$

Default: setting maximally optimistic $h(n) := 0$ is always admissible (gives Breadth-First-Search), but has no predictive power.

Note

Consider: formalization of a search problem

- often one needs to fulfil set of constraints
- which make calculating the optimal moves difficult

Idea:

- release one or more of these constraints
- the search problem then often becomes much easier
- ensuing h' for the less or unconstrained problem typically admissible for the original problem
- since removing the constraint does not increase the path length to the solution

Thus: releasing constraints on the original problem leads to admissible heuristics

(read up details on the heuristics for the 8-puzzle in Pearl and Russell/Norvig)

Mechanical Generation of Admissible Heuristics I

(Pearl 1984)

- consider **8-puzzle**
- move tile to adjacent empty tile
- until tiles all in order

- some start position

4	5	1
3	2	6
7		8

Mechanical Generation of Admissible Heuristics II

(Pearl 1984)

- consider **8-puzzle**
- move tile to adjacent empty tile
- until tiles all in order

- some start position
- ordered end position

1	2	3
4	5	6
7	8	

Mechanical Generation of Admissible Heuristics III

(Pearl 1984)

Formalize Idea:
- relax rules describing systems
- formalism to describe rules

Nilsson's STRIPS Formal Rule System

Preconditions: necessary predicates for invoking action

Additions: predicates to be added after action

Deletions: predicates no longer true after action

Mechanical Generation of Admissible Heuristics IV

(Pearl 1984)

Formalize Idea:
- relax rules describing systems
- formalism to describe rules

Nilsson's STRIPS Formal Rule System

Preconditions: necessary predicates for invoking action

Additions: predicates to be added after action

Deletions: predicates no longer true after action

For 8-Puzzle

ON(x, y)**:** tile x is on cell y

CLEAR(y)**:** cell y is clear of tiles

ADJ(y, z)**:** cell y is adjacent to cell z

Mechanical Generation of Admissible Heuristics V

(Pearl 1984)

State Description:

$\mathtt{ON}(X_1, C_1), \mathtt{ON}(X_2, C_2), \ldots, \mathtt{ON}(X_8, C_8), \mathtt{CLEAR}(C_9)$

Board Description:

$\mathtt{ADJ}(C_1, C_2), \mathtt{ADJ}(C_1, C_4), \ldots$

$\mathtt{MOVE}(x, y, z)$

Precondition: $\mathtt{ON}(x, y), \mathtt{CLEAR}(z), \mathtt{ADJ}(y, z)$

Add List: $\mathtt{ON}(x, z), \mathtt{CLEAR}(y)$

Delete List: $\mathtt{ON}(x, y), \mathtt{CLEAR}(z)$

We Seek

a sequence of $\mathtt{MOVE}(x, y, z)$ transforming initial state to a state satisfying the goal criteria

Modification: $\texttt{MOVE}(x, y, z)$

Precondition: $\texttt{ON}(x, y), \texttt{CLEAR}(z), \texttt{ADJ}(y, z)$

Add List: $\texttt{ON}(x, z), \texttt{CLEAR}(y)$

Delete List: $\texttt{ON}(x, y), \texttt{CLEAR}(z)$

Modification: $\mathtt{MOVE}'(x,y,z)$

Precondition: $\mathtt{ON}(x,y)$, $\cancel{\mathtt{CLEAR}(z), \mathtt{ADJ}(y,z)}$

Add List: $\mathtt{ON}(x,z), \mathtt{CLEAR}(y)$

Delete List: $\mathtt{ON}(x,y), \mathtt{CLEAR}(z)$

Relaxation

- delete $\mathtt{CLEAR}(z), \mathtt{ADJ}(y,z)$ from precondition
- i.e. a move now does not require a cell to be adjacent and empty
- successively "jump" tiles to target positions, "on top" of other tiles

Heuristic h_1

- number of misplaced tiles
- $h_1(s) = |\{x \mid \mathtt{ON}(x,y), x \neq y\}|$

  Here, s is the whole state (represented by the currently valid predicates). Note that x is implicitly a tile, and y a cell, via the predicate.

Heuristic h_1

- number of misplaced tiles
- $h_1(s) = |\{x \mid \mathtt{ON}(x,y), x \neq y\}|$

  Here, s is the whole state (represented by the currently valid predicates). Note that x is implicitly a tile, and y a cell, via the predicate.

4	5	1
3	2	6
7		8

Modification: $\texttt{MOVE}(x, y, z)$

Precondition: $\texttt{ON}(x, y), \texttt{CLEAR}(z), \texttt{ADJ}(y, z)$

Add List: $\texttt{ON}(x, z), \texttt{CLEAR}(y)$

Delete List: $\texttt{ON}(x, y), \texttt{CLEAR}(z)$

Modification: $\texttt{MOVE}''(x,y,z)$

Precondition: $\texttt{ON}(x,y), \cancel{\texttt{CLEAR}(z)}, \texttt{ADJ}(y,z)$

Add List: $\texttt{ON}(x,z), \texttt{CLEAR}(y)$

Delete List: $\texttt{ON}(x,y), \texttt{CLEAR}(z)$

Relaxation

- delete only $\texttt{CLEAR}(z)$ from precondition
- i.e. a move requires a cell to be adjacent, but not empty
- successively shift tiles to neighbouring positions, closer to target, but "on top" of other tiles

Heuristic h_2

- sum of number of steps to goal for each tile
- $h_2(s) = \sum_x ||\texttt{pos}(x) - \texttt{goal}(x)||_1$

`pos` is the position of the tile x and `goal` is where it should be. We interpret them as vector positions on a grid, so we can take the difference. $||.||_1$ is the *Manhattan metric*, i.e. the sum of vertical and horizontal steps to be taken for each tile to reach their goal.

Heuristic h_2

- sum of number of steps to goal for each tile
- $h_2(s) = \sum_x ||\texttt{pos}(x) - \texttt{goal}(x)||_1$

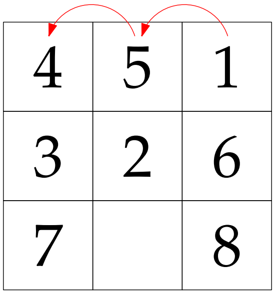
`pos` is the position of the tile x and `goal` is where it should be. We interpret them as vector positions on a grid, so we can take the difference. $||.||_1$ is the *Manhattan metric*, i.e. the sum of vertical and horizontal steps to be taken for each tile to reach their goal.

4	5	1
3	2	6
7		8

Heuristic h_2

- sum of number of steps to goal for each tile
- $h_2(s) = \sum_x ||\texttt{pos}(x) - \texttt{goal}(x)||_1$

`pos` is the position of the tile x and `goal` is where it should be. We interpret them as vector positions on a grid, so we can take the difference. $||.||_1$ is the *Manhattan metric*, i.e. the sum of vertical and horizontal steps to be taken for each tile to reach their goal.



Heuristic h_2

- sum of number of steps to goal for each tile
- $h_2(s) = \sum_x ||\texttt{pos}(x) - \texttt{goal}(x)||_1$

`pos` is the position of the tile x and `goal` is where it should be. We interpret them as vector positions on a grid, so we can take the difference. $||.||_1$ is the *Manhattan metric*, i.e. the sum of vertical and horizontal steps to be taken for each tile to reach their goal.

4	5	1
3	2	6
7		8

Mechanical Generation of Admissible Heuristics XVI

(Pearl 1984)

Heuristic h_2

- sum of number of steps to goal for each tile
- $h_2(s) = \sum_x ||\mathtt{pos}(x) - \mathtt{goal}(x)||_1$

`pos` is the position of the tile x and `goal` is where it should be. We interpret them as vector positions on a grid, so we can take the difference. $||.||_1$ is the *Manhattan metric*, i.e. the sum of vertical and horizontal steps to be taken for each tile to reach their goal.

4	5	1
3	2	6
7		8

Heuristic h_2

- sum of number of steps to goal for each tile
- $h_2(s) = \sum_x ||\mathtt{pos}(x) - \mathtt{goal}(x)||_1$

`pos` is the position of the tile x and `goal` is where it should be. We interpret them as vector positions on a grid, so we can take the difference. $||.||_1$ is the *Manhattan metric*, i.e. the sum of vertical and horizontal steps to be taken for each tile to reach their goal.

4	5	1
3	2	6
7		8

Heuristic h_2

- sum of number of steps to goal for each tile
- $h_2(s) = \sum_x ||\mathtt{pos}(x) - \mathtt{goal}(x)||_1$

`pos` is the position of the tile x and `goal` is where it should be. We interpret them as vector positions on a grid, so we can take the difference. $||.||_1$ is the *Manhattan metric*, i.e. the sum of vertical and horizontal steps to be taken for each tile to reach their goal.

4	5	1
3	2	6
7		8

Mechanical Generation of Admissible Heuristics XIX

(Pearl 1984)

Heuristic h_2

- sum of number of steps to goal for each tile
- $h_2(s) = \sum_x ||\mathtt{pos}(x) - \mathtt{goal}(x)||_1$

`pos` is the position of the tile x and `goal` is where it should be. We interpret them as vector positions on a grid, so we can take the difference. $||.||_1$ is the *Manhattan metric*, i.e. the sum of vertical and horizontal steps to be taken for each tile to reach their goal.

4	5	1
3	2	6
7		8

Heuristic h_2

- sum of number of steps to goal for each tile
- $h_2(s) = \sum_x ||\mathtt{pos}(x) - \mathtt{goal}(x)||_1$

`pos` is the position of the tile x and `goal` is where it should be. We interpret them as vector positions on a grid, so we can take the difference. $||.||_1$ is the *Manhattan metric*, i.e. the sum of vertical and horizontal steps to be taken for each tile to reach their goal.

4	5	1
3	2	6
7		8

Mechanical Generation of Admissible Heuristics XXI

(Pearl 1984)

Heuristic h_2

- sum of number of steps to goal for each tile
- $h_2(s) = \sum_x ||\mathtt{pos}(x) - \mathtt{goal}(x)||_1$

`pos` is the position of the tile x and `goal` is where it should be. We interpret them as vector positions on a grid, so we can take the difference. $||.||_1$ is the *Manhattan metric*, i.e. the sum of vertical and horizontal steps to be taken for each tile to reach their goal.

4	5	1
3	2	6
7		8

Heuristic h_2

- sum of number of steps to goal for each tile
- $h_2(s) = \sum_x ||\mathtt{pos}(x) - \mathtt{goal}(x)||_1$

`pos` is the position of the tile x and `goal` is where it should be. We interpret them as vector positions on a grid, so we can take the difference. $||.||_1$ is the *Manhattan metric*, i.e. the sum of vertical and horizontal steps to be taken for each tile to reach their goal.

4	5	1
3	2	6
7		8

Mechanical Generation of Admissible Heuristics XXIII

(Pearl 1984)

Heuristic h_2

- sum of number of steps to goal for each tile
- $h_2(s) = \sum_x ||\texttt{pos}(x) - \texttt{goal}(x)||_1$

`pos` is the position of the tile x and `goal` is where it should be. We interpret them as vector positions on a grid, so we can take the difference. $||.||_1$ is the *Manhattan metric*, i.e. the sum of vertical and horizontal steps to be taken for each tile to reach their goal.

4	5	1
3	2	6
7		8

Modification: $\texttt{MOVE}(x, y, z)$

Precondition: $\texttt{ON}(x, y), \texttt{CLEAR}(z), \texttt{ADJ}(y, z)$

Add List: $\texttt{ON}(x, z), \texttt{CLEAR}(y)$

Delete List: $\texttt{ON}(x, y), \texttt{CLEAR}(z)$

4	5	1
3	2	6
7		8

Modification: $\mathtt{MOVE}'''(x,y,z)$

Precondition: $\mathtt{ON}(x,y), \mathtt{CLEAR}(z), \cancel{\mathtt{ADJ}(y,z)}$

Add List: $\mathtt{ON}(x,z), \mathtt{CLEAR}(y)$

Delete List: $\mathtt{ON}(x,y), \mathtt{CLEAR}(z)$

4	5	1
3	2	6
7		8

Relaxation

- delete only $\mathtt{ADJ}(y,z)$ from precondition
- i.e. a move requires a cell to be empty, but not adjacent
- successively jump tiles to empty position if target. If empty position itself is not a target (i.e. should remain empty), jump random tile

Modification: $\texttt{MOVE}'''(x, y, z)$

Precondition: $\texttt{ON}(x, y), \texttt{CLEAR}(z), \cancel{\texttt{ADJ}(y, z)}$

Add List: $\texttt{ON}(x, z), \texttt{CLEAR}(y)$

Delete List: $\texttt{ON}(x, y), \texttt{CLEAR}(z)$

4	5	1
3	2	6
7		8

Relaxation

- delete only $\texttt{ADJ}(y, z)$ from precondition
- i.e. a move requires a cell to be empty, but not adjacent
- successively jump tiles to empty position if target. If empty position itself is not a target (i.e. should remain empty), jump random tile

This is not an obvious strategy and was discovered much later than the others! (Gaschnig 1979)

Optimization of Existing Heuristics

- given admissible heuristics $h_1, h_2, h_3, \ldots, h_n$:
- is it possible to construct a heuristic at least as good as any of $h_1, h_2, h_3, \ldots, h_n$?

Heuristic Construction XXVII

Construction of Heuristics: Given admissible heuristics $h_1, h_2, h_3, \ldots, h_n$, is it possible to construct a heuristic that is as least as good as any of the above?

Consideration: first, what is a good heuristics?

Note: clearly a maximally admissible $h = 0$ is a bad heuristics, not helping at all

Ergo: a heuristic is most expressive/helpful/good the *larger* it is!

Heuristic Construction XXVIII

Construction of Heuristics: Given admissible heuristics $h_1, h_2, h_3, \ldots, h_n$, is it possible to construct a heuristic that is as least as good as any of the above?

Consideration: first, what is a good heuristics?

Note: clearly a maximally admissible $h = 0$ is a bad heuristics, not helping at all

Ergo: a heuristic is most expressive/helpful/good the *larger* it is!

Bottom Line: the heuristics

$$h_{\mathsf{max}} := \max(h_1, h_2, h_3, \ldots, h_n)$$

is at least as good as any of the h_i.

Heuristic Construction XXIX

Construction of Heuristics: Given admissible heuristics $h_1, h_2, h_3, \ldots, h_n$, is it possible to construct a heuristic that is as least as good as any of the above?

Consideration: first, what is a good heuristics?

Note: clearly a maximally admissible $h = 0$ is a bad heuristics, not helping at all

Ergo: a heuristic is most expressive/helpful/good the *larger* it is!

Bottom Line: the heuristics

$$h_{\mathsf{max}} := \max(h_1, h_2, h_3, \ldots, h_n)$$

is at least as good as any of the h_i.

Note: the best possible heuristic were the true value of the cost to a goal if it were known (which, in general, is difficult).

Theory and Practice of Artificial Intelligence

Games

Daniel Polani

School of Computer Science
University of Hertfordshire

March 9, 2017

Games

More Precisely:

- two-person (not multi-person; no gang-ups)
- perfect information (no card games)
- deterministic (no backgammon)
- alternating moves (no rock/scissors/paper)
- zero-sum (no prisoner's dilemma)

games

Game Structure

Conditions: game is over when *terminal* position reached where game ends (no successor moves).

Possible Outcomes: consider *win/loss/draw*. Other, intermediate outcomes also possible.

Game State Structure

Game:

- game position
- terminal won position
- terminal lost
- non-terminal won
- *us-to-move* (player A)
- *them-to-move* (player B)

Position Utilities

Motivation: since, in general, game trees are too big to be completely solved, use a *utility* (value) function to indicate which positions are more promising than another.

Implication: quality of a game state characterized by its value (utility) U, a real-valued number

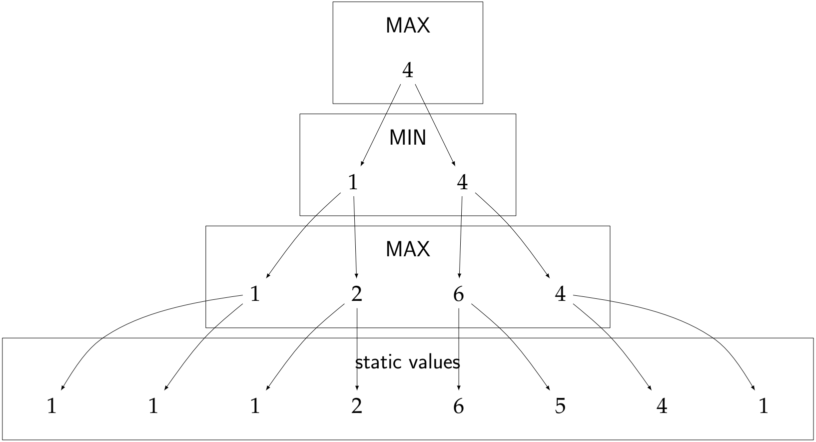
Note: "promising" subtrees are indicated by a high value of U for starting states.

Position Utilities II

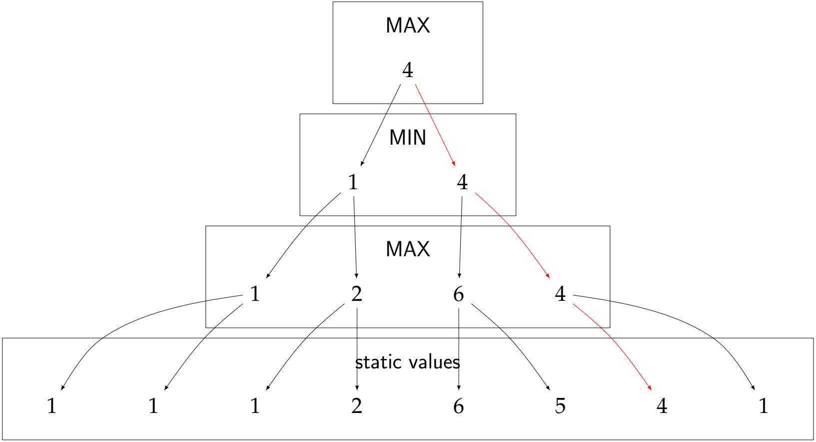
Note: the *true* value U of a position indicates the state of the position won/lost/draw, e.g.

- $U = 100$**:** current position allows player A to win (on optimal game from both sides)
- $U = -100$**:** current position is lost for player A (on optimal game from both sides)
- $U = 0$**:** position is a draw (no player can force a win)

Minimax Principle



Minimax Principle (Main Variation)



Minimax view of utilities

Consider: $U(P)$, the utility of a position

Let: $S(P) = \{P_1, P_2, \ldots, P_n\}$ be the set of successors for position P

Minimax Utility: define

$$U(P) = \begin{cases} U_{\mathsf{static}}(P) & \text{if } P \text{ terminal, i.e. } S(P) = \{\} \\ \max\limits_{P_i \in S(P)} U(P_i) & \text{if } P \text{ is a MAX-to-move position} \\ \min\limits_{P_i \in S(P)} U(P_i) & \text{if } P \text{ is a MIN-to-move position} \end{cases}$$

The Alpha-Beta Algorithm

Observation:

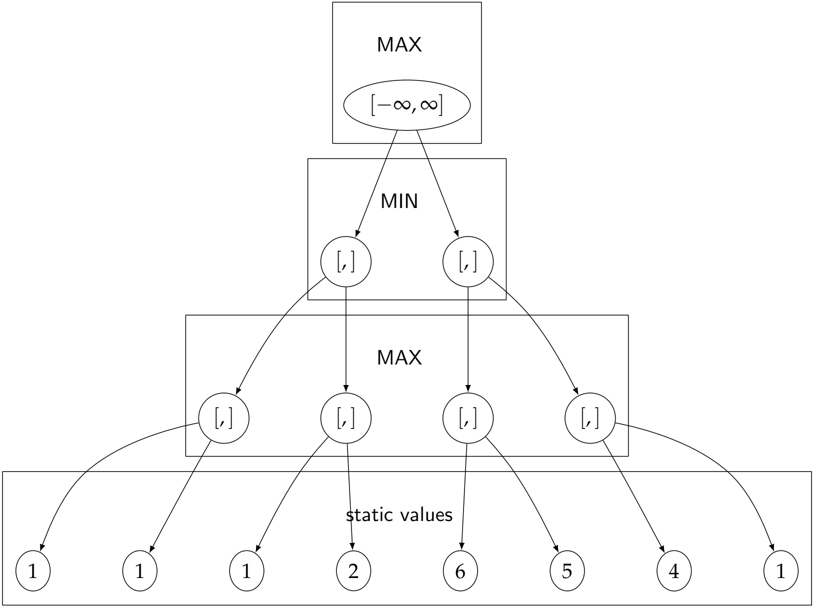
- sometimes we know a move is not good and will never be covered
- in that case, the exact utility of the node is not needed

α-β **principle:**

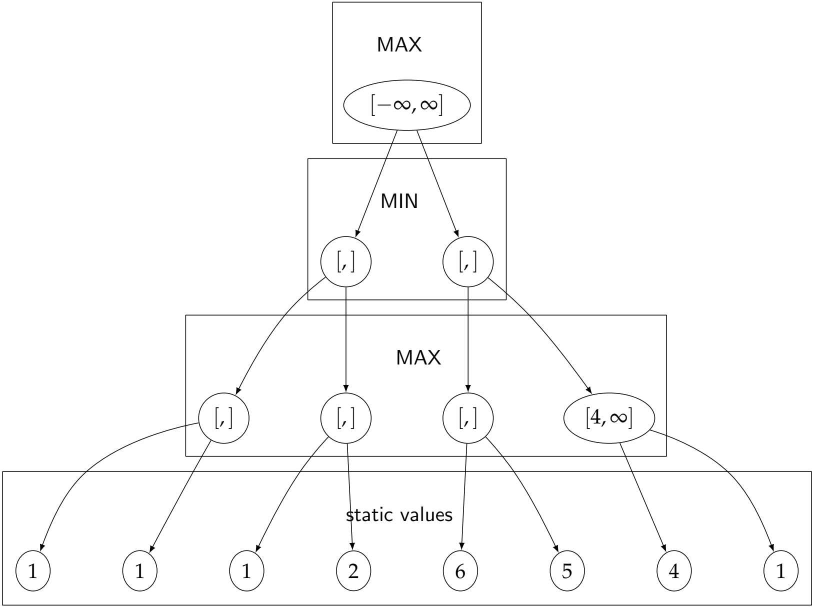
- search for the utility of a position but only if in the interval $[\alpha, \beta]$
- if it is outside, its exact value is not important, we will be prevented from taking that path anyway

Illustration: see following slides

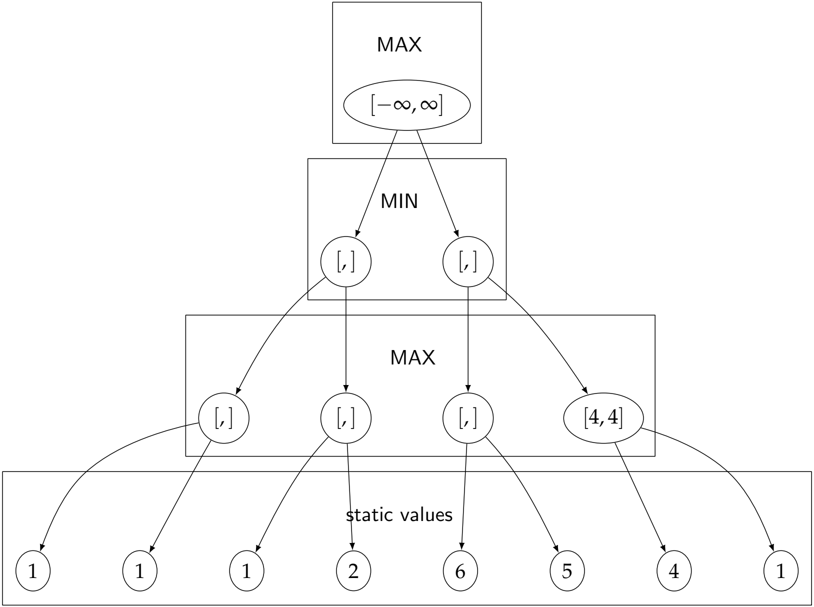
The Alpha-Beta Algorithm



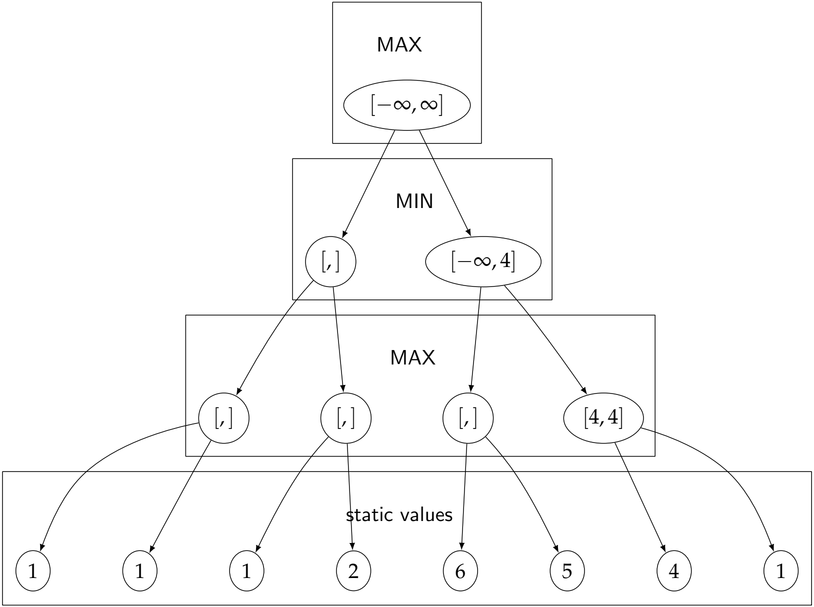
The Alpha-Beta Algorithm



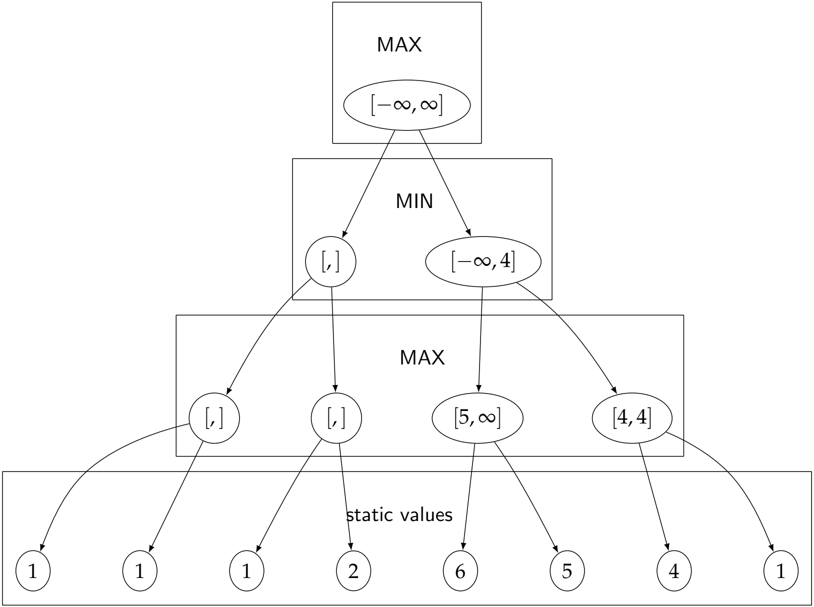
The Alpha-Beta Algorithm



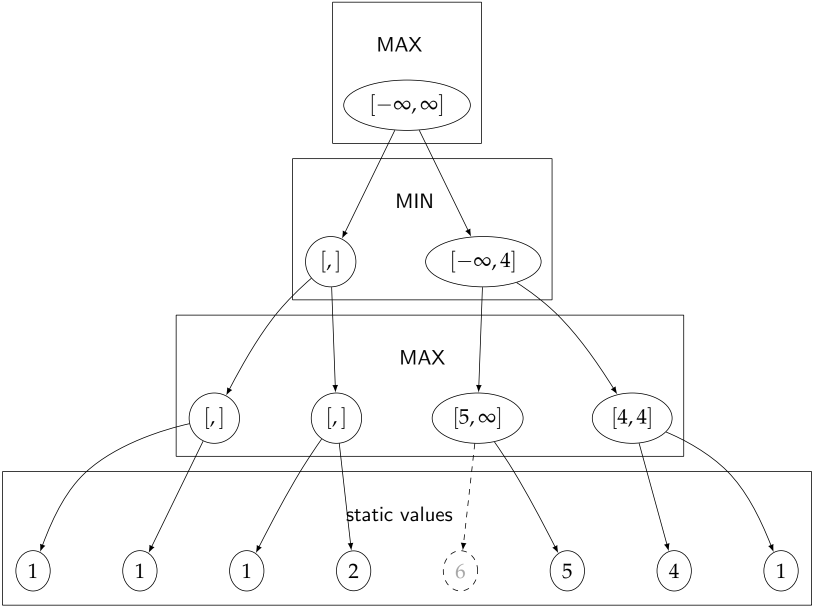
The Alpha-Beta Algorithm



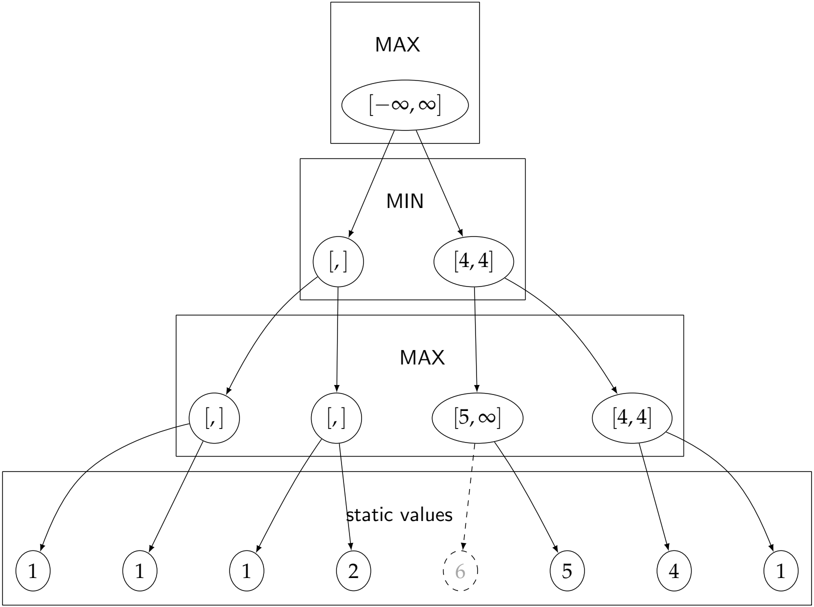
The Alpha-Beta Algorithm



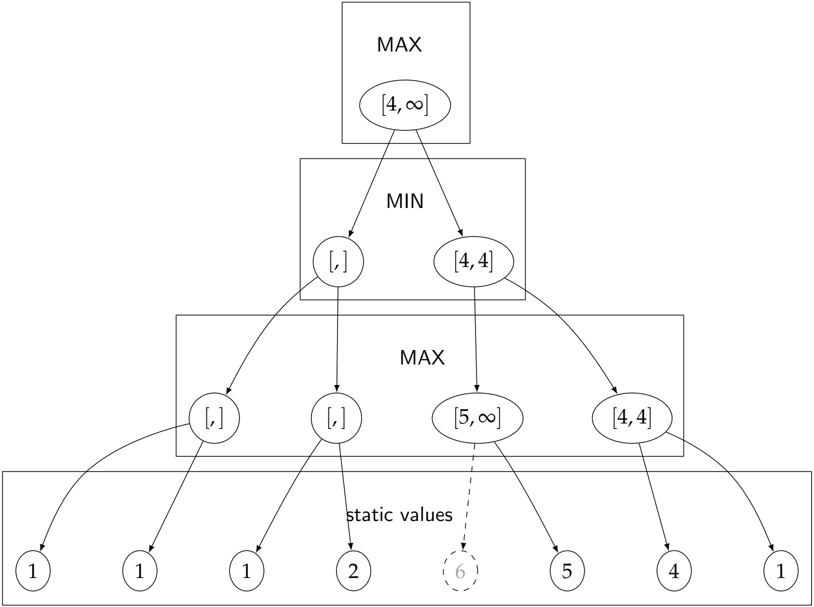
The Alpha-Beta Algorithm



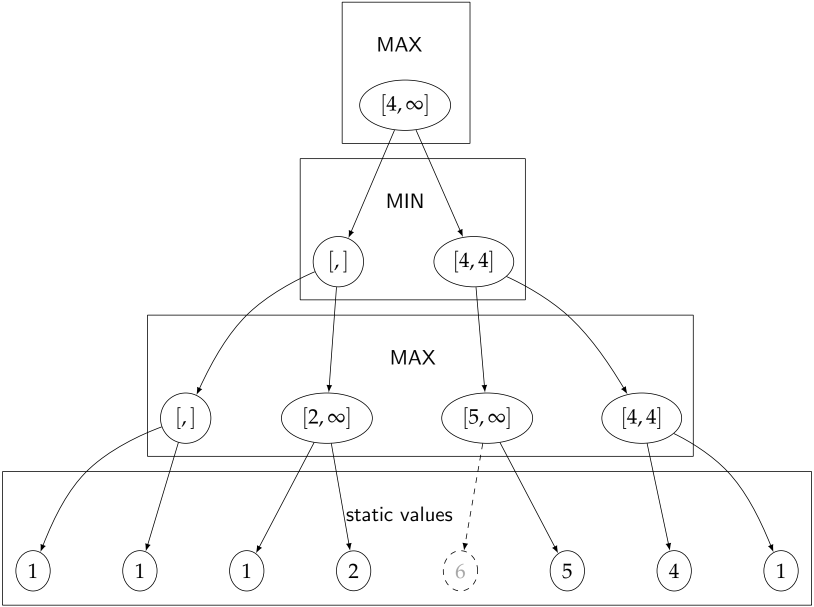
The Alpha-Beta Algorithm



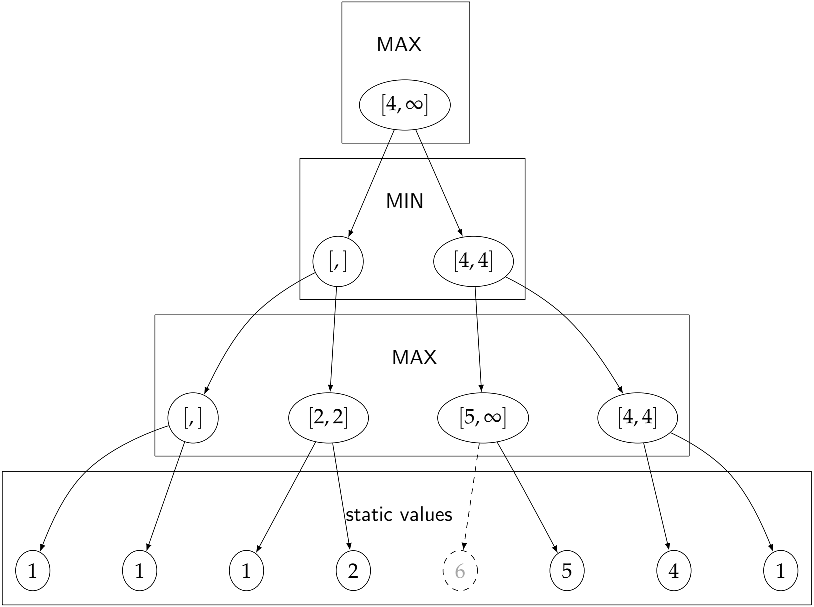
The Alpha-Beta Algorithm



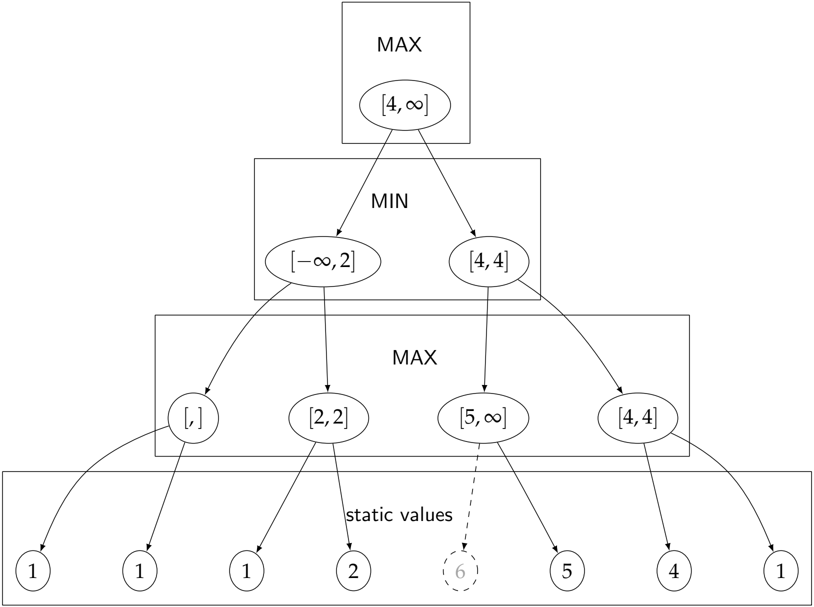
The Alpha-Beta Algorithm



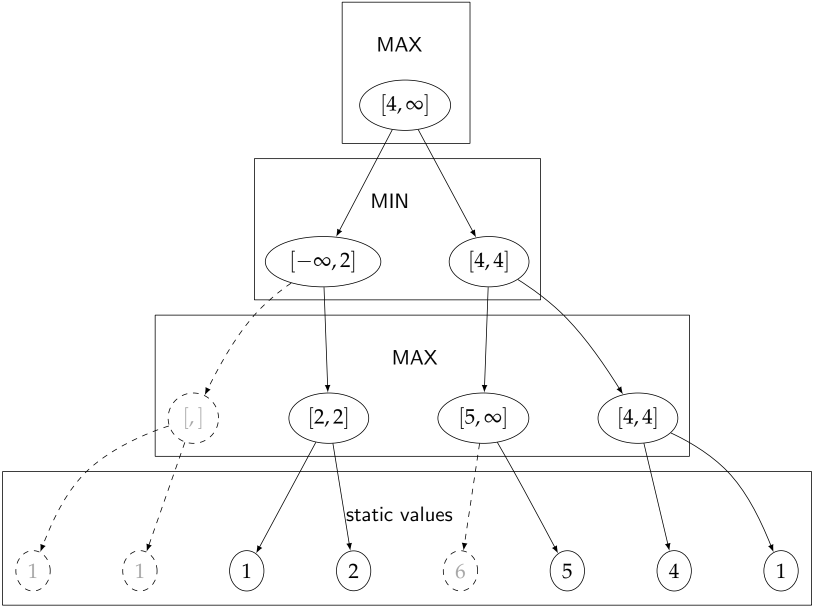
The Alpha-Beta Algorithm



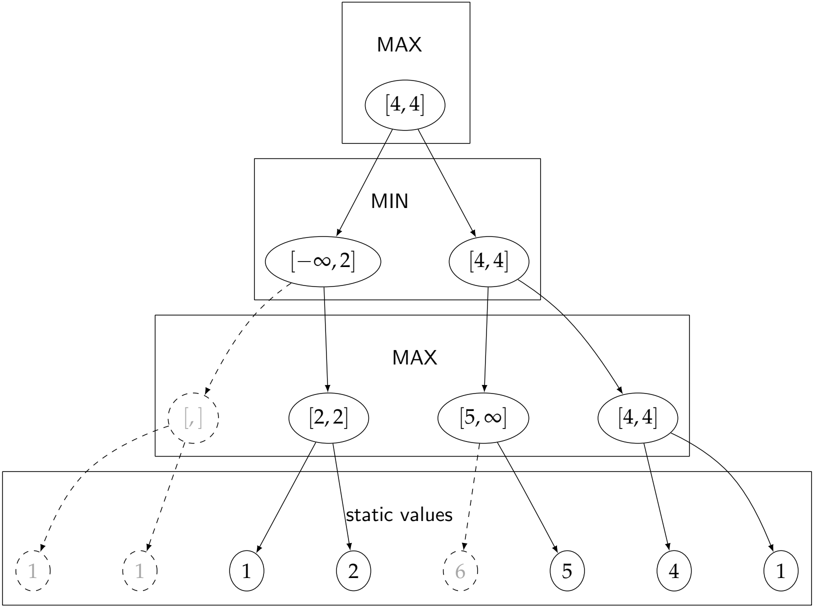
The Alpha-Beta Algorithm



The Alpha-Beta Algorithm



The Alpha-Beta Algorithm



Alpha-Beta Algorithm: Properties

α: worst guaranteed utility for MAX
(and best achievable value for MIN)

β: worst guaranteed utility for MIN
(and best achievable value for MAX)

Good Enough Utility: a utility $U(P, \alpha, \beta)$ is a utility such that

$$\begin{aligned} U(P,\alpha,\beta) &< \alpha & \text{if } U(P) < \alpha \\ U(P,\alpha,\beta) &= U(P) & \text{if } \alpha \leq U(P) \leq \beta \\ U(P,\alpha,\beta) &> \beta & \text{if } U(P) > \beta\,. \end{aligned}$$

In Particular: $U(P, -\infty, \infty) = U(P)$

Remark: in the best case, this reduces the search branching factor from b for minimax to $\sqrt{b}$

Thus: can search twice as deeply as with minimax with the same evaluation effort

Further Improvements

1. limitation of move selection
2. heuristic value function (cutoff before final state)
3. quiescence heuristics

Further Improvements

1. limitation of move selection
2. heuristic value function (cutoff before final state)
3. quiescence heuristics
4. **endgame algorithm**

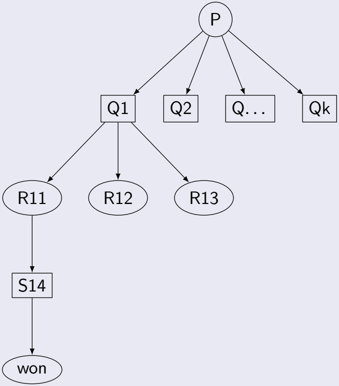
Further Improvements

1. limitation of move selection
2. heuristic value function (cutoff before final state)
3. quiescence heuristics
4. **endgame algorithm**
5. **UCT Monte Carlo Tree Search**

Game-Playing to the End: Idea

End Games: consider

- game with only *win/loss*
- 2 players **us** and **them**
- playing alternatively
- solution: win for **us**



Interpretation: game is won if solution tree exists, i.e. tree begins with an

us node: there is a choice for **us** leading to an

them node: such that all possible choices for **them** lead to an

us node: and so on until

Goal: successful solution (**win**) is found

Interpretation

It means: **us** has won (solution tree) if it is either

- in a **winning position** or it can always choose a move leading
- to a **losing** position of **them**; i.e. a position such that all moves that **them** can choose lead
- to a winning position of **us** (i.e. again to a solution tree).

Note: **us** does not have to have a solution tree. Either

- **them** could have a solution tree (in which **us** loses)
- or neither of them have, so none of the players can force a win.

Yes, I treat **us** as singular player and not as *pluralis majestatis*.

Endgame Algorithm

Endgame Algorithm: for **us**

1. consider final (0-step) winning positions for **us**
2. compute 1-step losing positions for **them**, i.e.
   - all positions for **them** from which
   - **all** immediate successors lead
   - to a 0-step winning position for **us**
3. compute 2-step winning positions for **us**, i.e.
   - all positions where **us** can choose
   - *one* immediate successor to lead
   - to a 1-step losing position for **them**
4. compute 3-step losing positions for **them**, i.e.
   - all positions for **them** where
   - *all* successors lead
   - to a less-than-3 (i.e. 2- or 0-) winning position for **us**.
5. and so on, until no more new positions are collected or maximum depth are exhausted

Result: if no maximum depth limit, the final outcome is a

- list of winning positions for **us** (with maximum depths)
- a list of losing positions for **them** (with maximum depths)
- and a list of tied positions

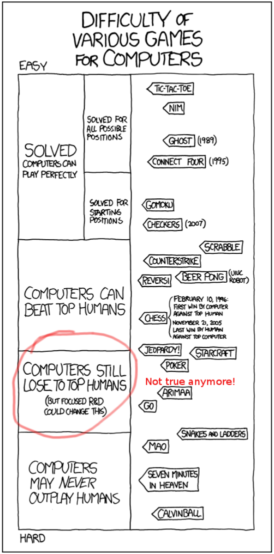
Theory and Practice of Artificial Intelligence

Further Games

Daniel Polani

School of Computer Science
University of Hertfordshire

March 9, 2017



`https://xkcd.com/1002/` (CC BY-NC 2.5)

UCT Monte Carlo Tree Search I

- one of the great breakthroughs in game AIs
- based on exploration/exploitation tradeoffs regret (Auer 2003)
- generalized to trees (Kocsis and Szepesvári 2006)

Note: do not have the time for the full theory
just sketch the method

UCT Monte Carlo Tree Search II

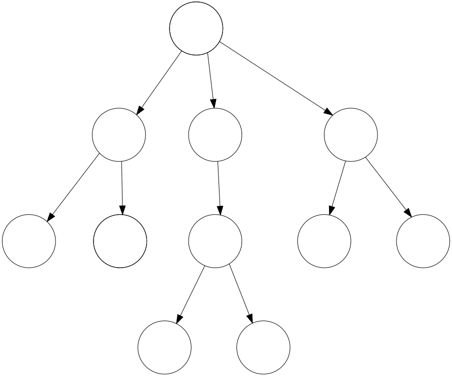
(Browne 2012; Browne et al. 2012; Bradberry 2015)

Outset:
- consider an **already expanded partial tree**
- assume every node contains a
  - sum of rewards $\sum V_i$ hitherto collected from nodes beneath it
  - number of runs n that went through that node

for now, just a search, will generalize to games later

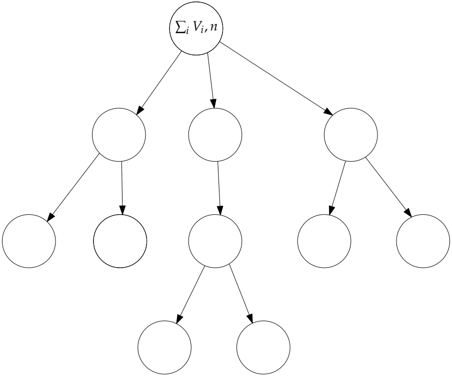
UCT Monte Carlo Tree Search III

(Browne 2012; Browne et al. 2012; Bradberry 2015)



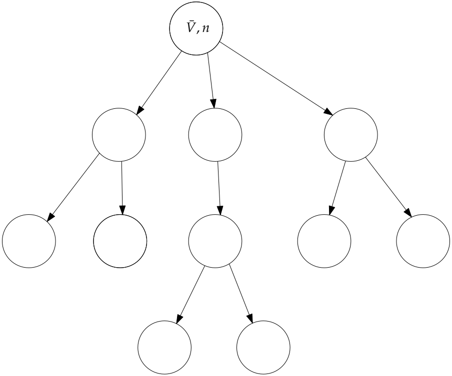
UCT Monte Carlo Tree Search IV

(Browne 2012; Browne et al. 2012; Bradberry 2015)



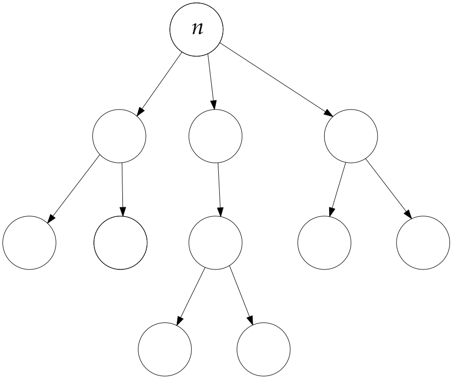
UCT Monte Carlo Tree Search V

(Browne 2012; Browne et al. 2012; Bradberry 2015)



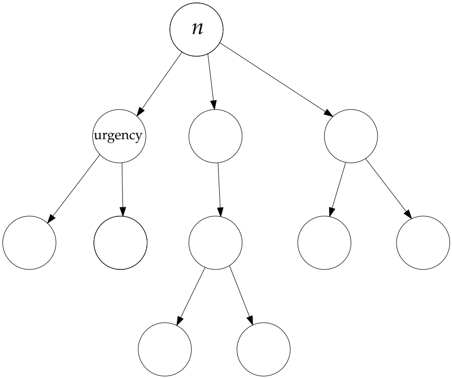
UCT Monte Carlo Tree Search VI

(Browne 2012; Browne et al. 2012; Bradberry 2015)



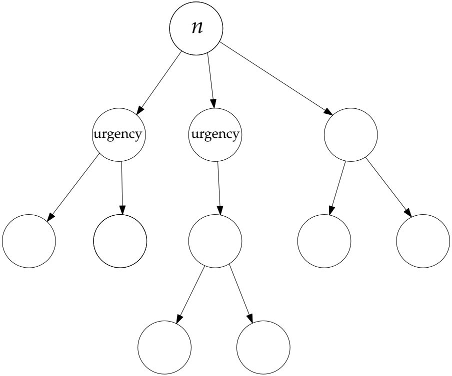
UCT Monte Carlo Tree Search VII

(Browne 2012; Browne et al. 2012; Bradberry 2015)



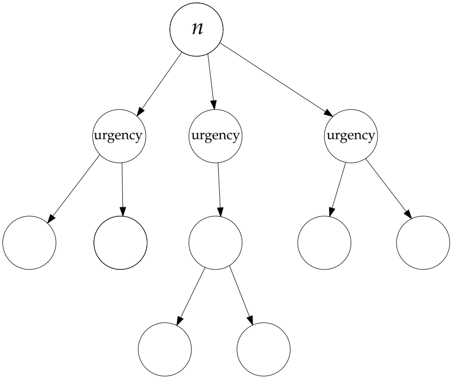
UCT Monte Carlo Tree Search VIII

(Browne 2012; Browne et al. 2012; Bradberry 2015)



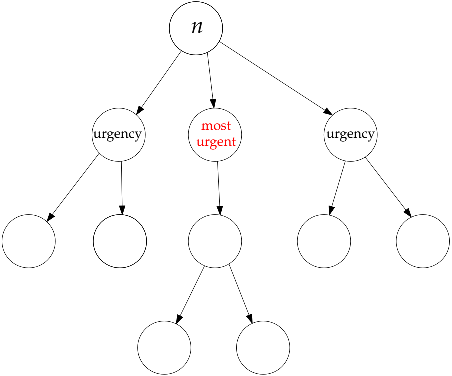
UCT Monte Carlo Tree Search IX

(Browne 2012; Browne et al. 2012; Bradberry 2015)



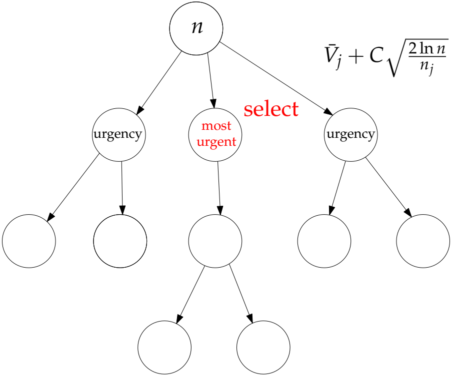
UCT Monte Carlo Tree Search X

(Browne 2012; Browne et al. 2012; Bradberry 2015)



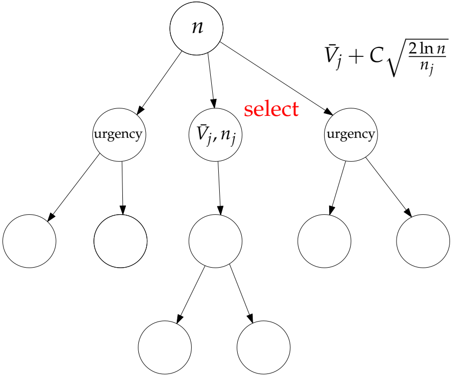
UCT Monte Carlo Tree Search XI

(Browne 2012; Browne et al. 2012; Bradberry 2015)



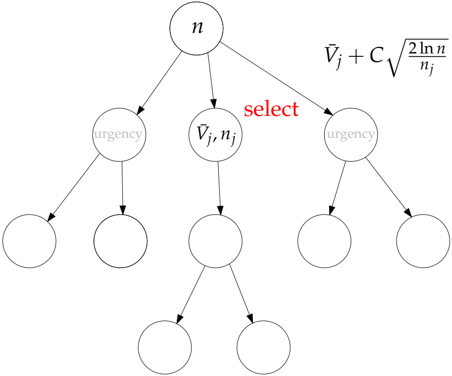
UCT Monte Carlo Tree Search XII

(Browne 2012; Browne et al. 2012; Bradberry 2015)



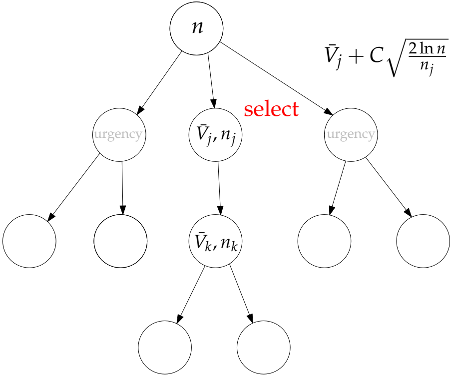
UCT Monte Carlo Tree Search XIII

(Browne 2012; Browne et al. 2012; Bradberry 2015)



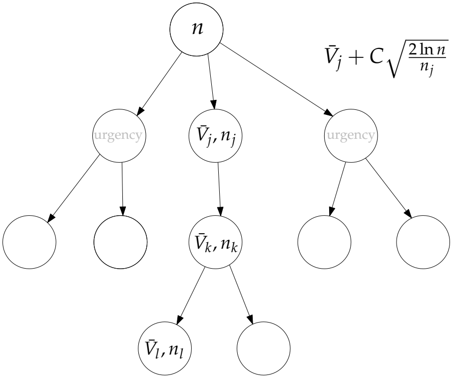
UCT Monte Carlo Tree Search XIV

(Browne 2012; Browne et al. 2012; Bradberry 2015)



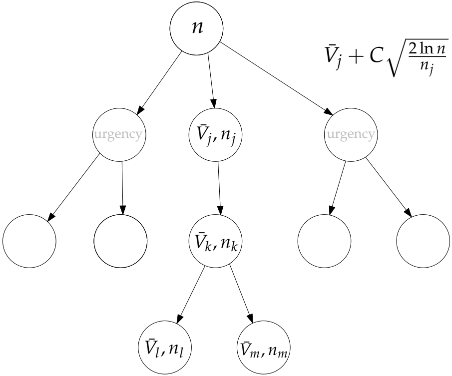
UCT Monte Carlo Tree Search XV

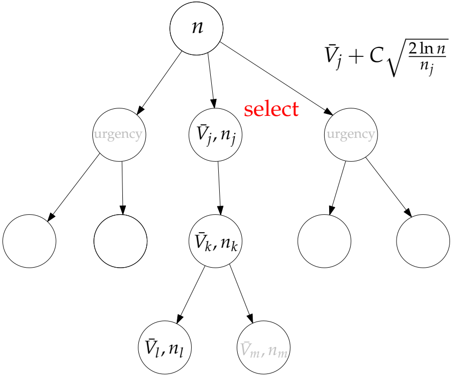
(Browne 2012; Browne et al. 2012; Bradberry 2015)

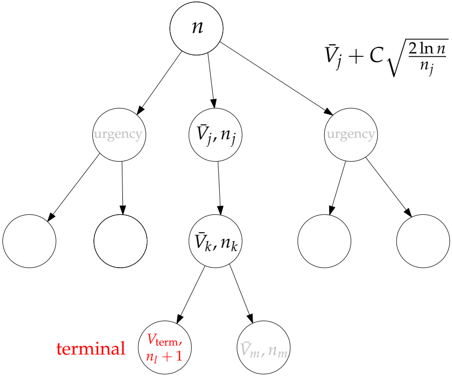


UCT Monte Carlo Tree Search XVI

(Browne 2012; Browne et al. 2012; Bradberry 2015)

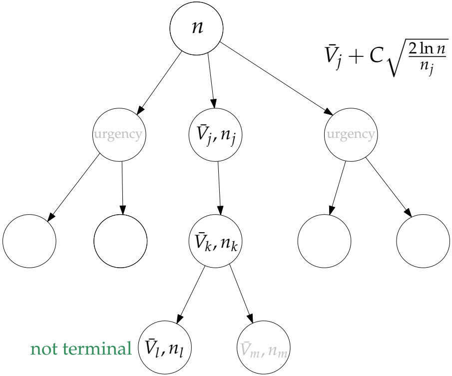



n
$\bar{V}_j + C\sqrt{\frac{2\ln n}{n_j}}$
select
urgency
$\bar{V}_j, n_j$
urgency
$\bar{V}_k, n_k$
$\bar{V}_l, n_l$
$\bar{V}_m, n_m$


n
$\bar{V}_j + C\sqrt{\frac{2\ln n}{n_j}}$
urgency
$\bar{V}_j, n_j$
urgency
$\bar{V}_k, n_k$
terminal
V_{term}, $n_l + 1$
$\bar{V}_m, n_m$

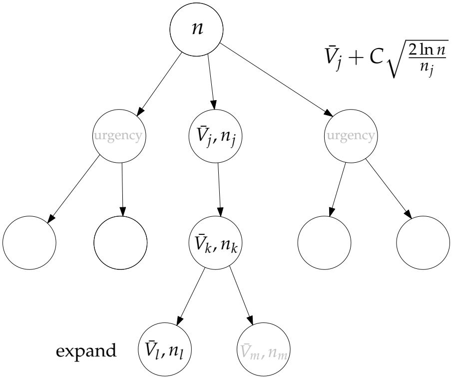
UCT Monte Carlo Tree Search XIX

(Browne 2012; Browne et al. 2012; Bradberry 2015)



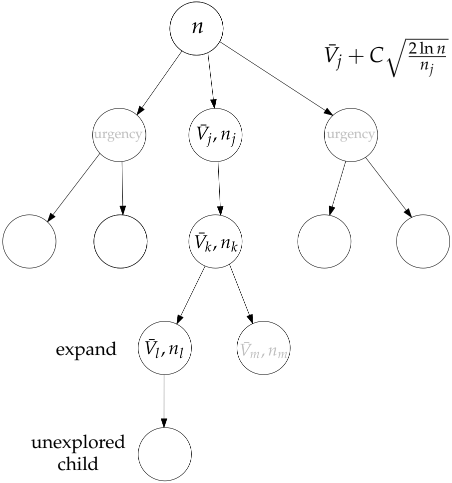
UCT Monte Carlo Tree Search XX

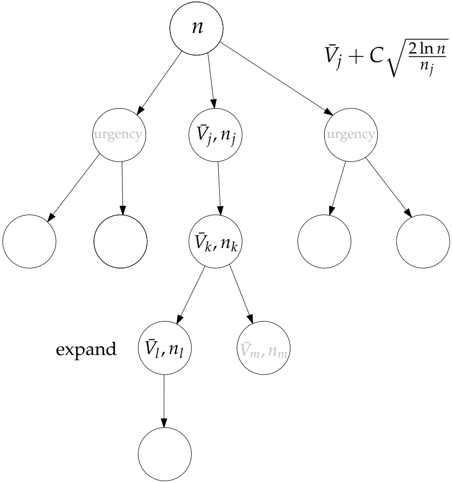
(Browne 2012; Browne et al. 2012; Bradberry 2015)



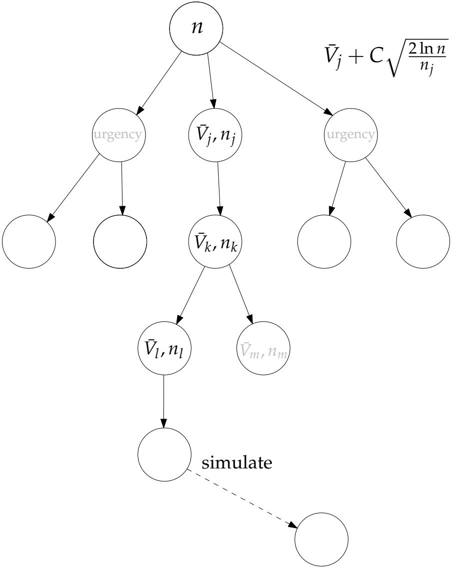
UCT Monte Carlo Tree Search XXI

(Browne 2012; Browne et al. 2012; Bradberry 2015)



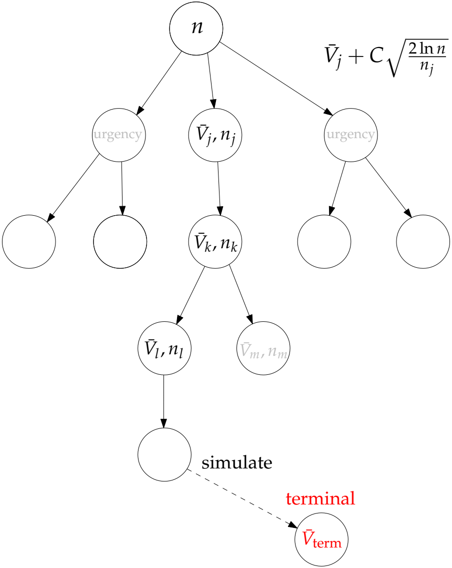


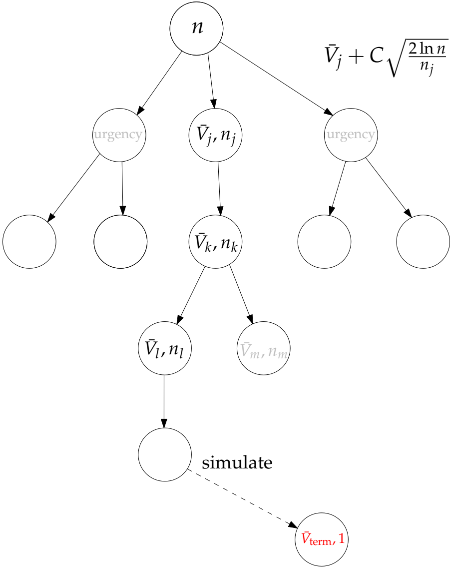
n
$\bar{V}_j + C\sqrt{\frac{2\ln n}{n_j}}$
urgency
$\bar{V}_j, n_j$
urgency
$\bar{V}_k, n_k$
expand
$\bar{V}_l, n_l$
$\bar{V}_m, n_m$


n
$\bar{V}_j + C\sqrt{\frac{2\ln n}{n_j}}$
urgency
$\bar{V}_j, n_j$
urgency
$\bar{V}_k, n_k$
$\bar{V}_l, n_l$
$\bar{V}_m, n_m$
simulate

UCT Monte Carlo Tree Search XXIV

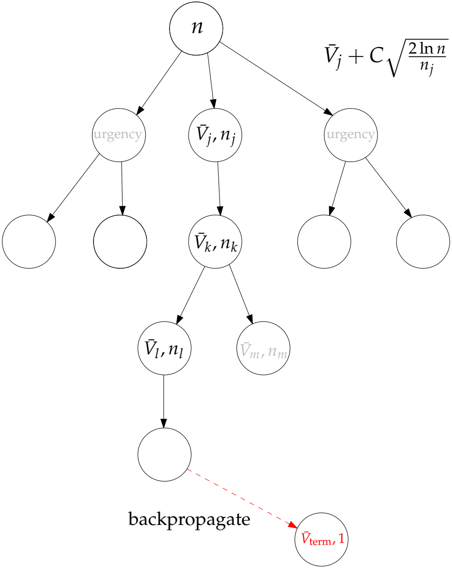
(Browne 2012; Browne et al. 2012; Bradberry 2015)

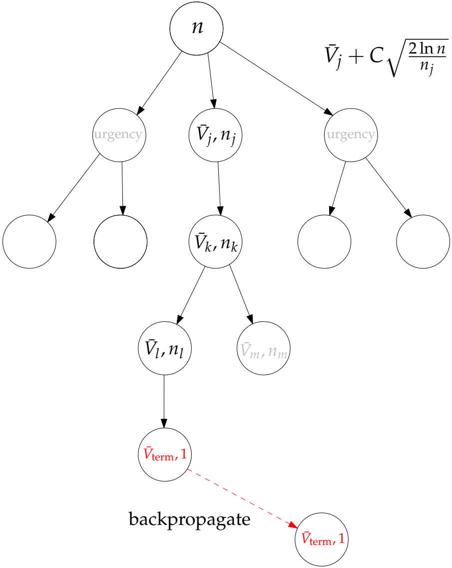


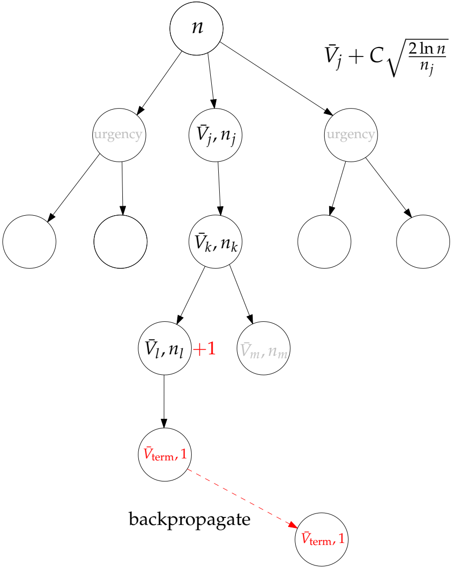

n
$\bar{V}_j + C\sqrt{\frac{2\ln n}{n_j}}$
urgency
$\bar{V}_j, n_j$
urgency
$\bar{V}_k, n_k$
$\bar{V}_l, n_l$
$\bar{V}_m, n_m$
simulate
$\bar{V}_{\text{term}}, 1$

UCT Monte Carlo Tree Search XXVI

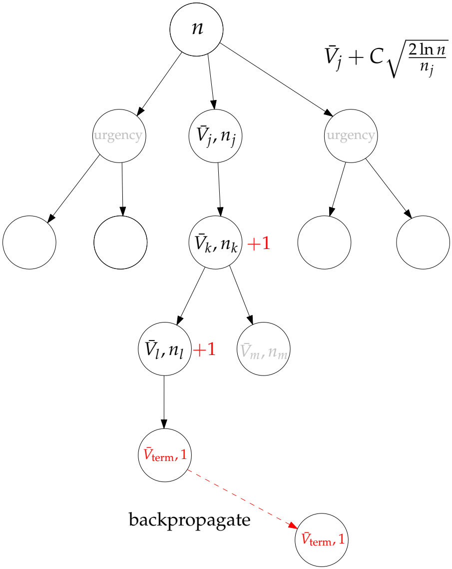
(Browne 2012; Browne et al. 2012; Bradberry 2015)




n
$\bar{V}_j + C\sqrt{\frac{2\ln n}{n_j}}$
urgency
$\bar{V}_j, n_j$
urgency
$\bar{V}_k, n_k$
$\bar{V}_l, n_l$
$\bar{V}_m, n_m$
$\bar{V}_{\text{term}}, 1$
backpropagate
$\bar{V}_{\text{term}}, 1$

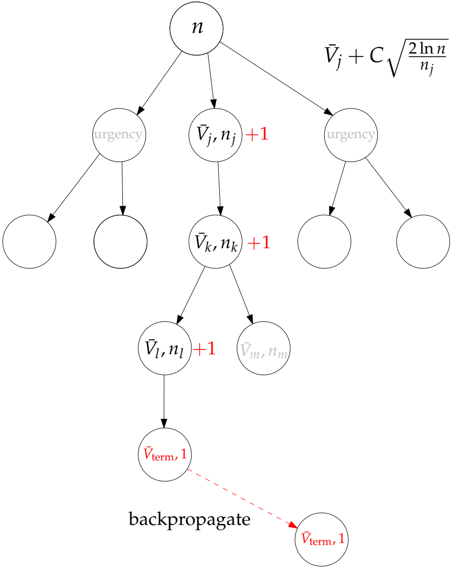


n
$\bar{V}_j + C\sqrt{\frac{2\ln n}{n_j}}$
urgency
$\bar{V}_j, n_j$
urgency
$\bar{V}_k, n_k$
$\bar{V}_l, n_l$
+1
$\bar{V}_m, n_m$
$\bar{V}_{\text{term}}, 1$
backpropagate
$\bar{V}_{\text{term}}, 1$


n
$\bar{V}_j + C\sqrt{\frac{2\ln n}{n_j}}$
urgency
$\bar{V}_j, n_j$
urgency
$\bar{V}_k, n_k$ +1
$\bar{V}_l, n_l$ +1
$\bar{V}_m, n_m$
$\bar{V}_{\text{term}}, 1$
backpropagate
$\bar{V}_{\text{term}}, 1$

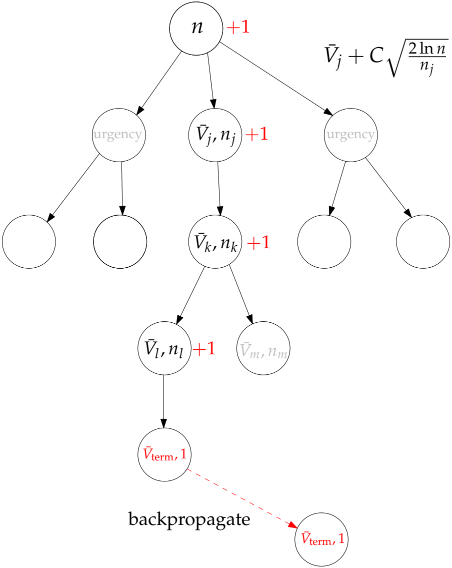
UCT Monte Carlo Tree Search XXX

(Browne 2012; Browne et al. 2012; Bradberry 2015)

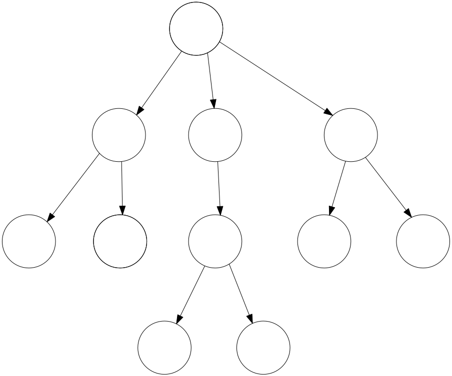


UCT Monte Carlo Tree Search XXXI

(Browne 2012; Browne et al. 2012; Bradberry 2015)

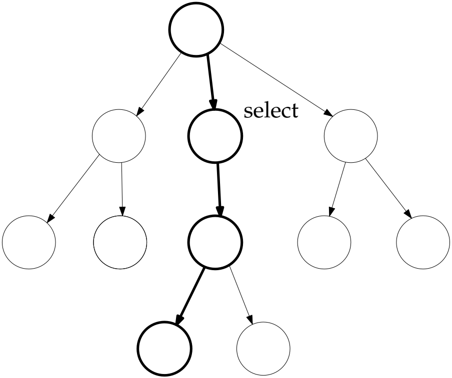


Summary



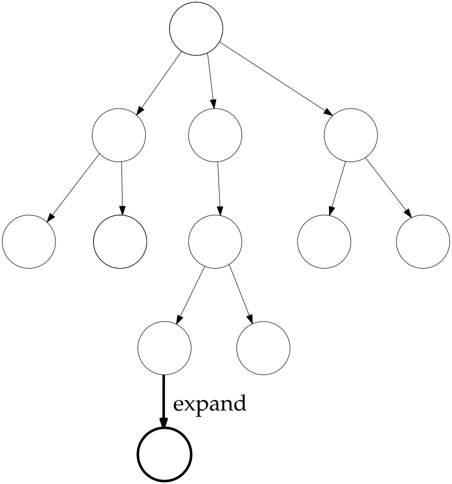
Summary

1. select



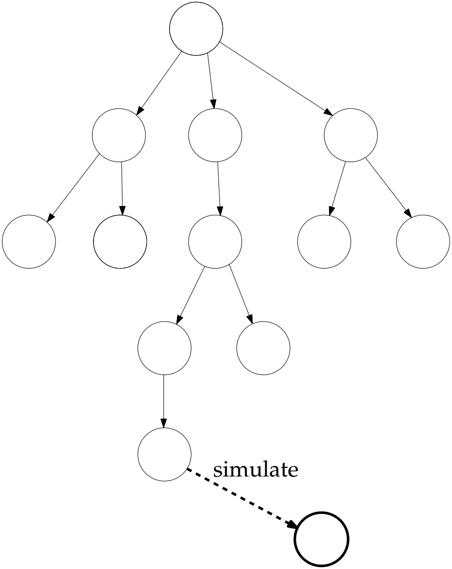
Summary

1. select
2. expand



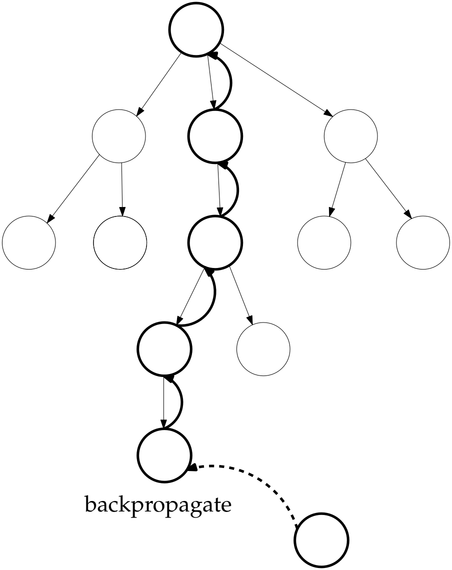
Summary

1. select
2. expand
3. simulate



Summary

1. select
2. expand
3. simulate
4. backpropagate



Additional Comments

Note:
- we treated it as a puzzle problem
- rewards just positive

But:
- in a game, antagonistic situation
- either: use NEG-MAX picture
- turn reward around at each step (multiply by -1 for each level)
  (Browne 2012)
- or: have utility for the player of the particular incremented if they won the game

Mystery Factor: Urgency

Confidence Bound

- consider a sequence of random rewards (value payoffs)
- with mean $\bar{V}$
- it is not perfectly accurate
- from Hoeffding's inequality (google it if you dare!), one gets that the true mean is "with good probability" in an interval

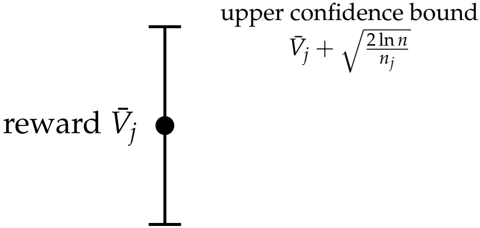
$$\left[\bar{V}_j - \sqrt{\frac{2\ln n}{n_j}}, \bar{V}_j + \sqrt{\frac{2\ln n}{n_j}}\right]$$

  if option j is visited n_j times and n total runs have been made
- it can be shown that selecting the branch with highest **upper confidence bound** (UCB)

$$\bar{V}_j + \sqrt{\frac{2\ln n}{n_j}}$$

  minimizes regret asymptotically

(Auer 2003; Kocsis and Szepesvári 2006)



$$\bar{V}_j + \sqrt{\frac{2 \ln n}{n_j}}$$

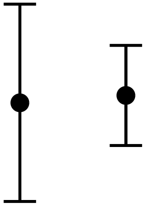
Criterion

(Browne 2012; Browne et al. 2012)

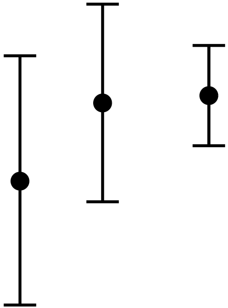
$$\bar{V}_j + \sqrt{\frac{2 \ln n}{n_j}}$$

Criterion

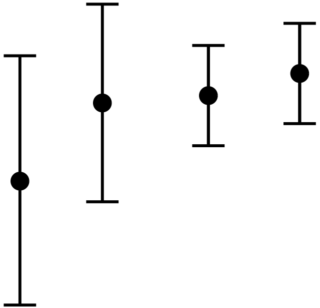
(Browne 2012; Browne et al. 2012)



$$\bar{V}_j + \sqrt{\frac{2 \ln n}{n_j}}$$

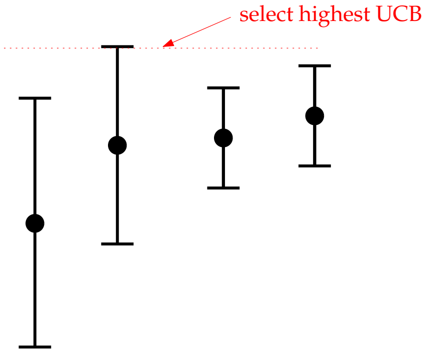


$$\bar{V}_j + \sqrt{\frac{2 \ln n}{n_j}}$$

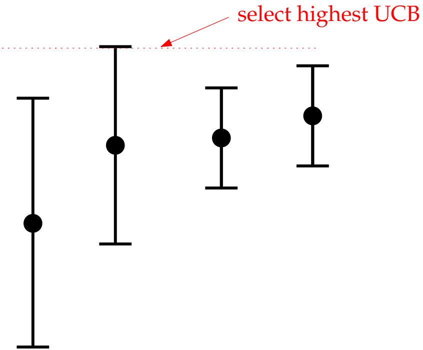


$$\bar{V}_j + \sqrt{\frac{2 \ln n}{n_j}}$$

Criterion
(Browne 2012; Browne et al. 2012)



$$\bar{V}_j + \sqrt{\frac{2 \ln n}{n_j}}$$

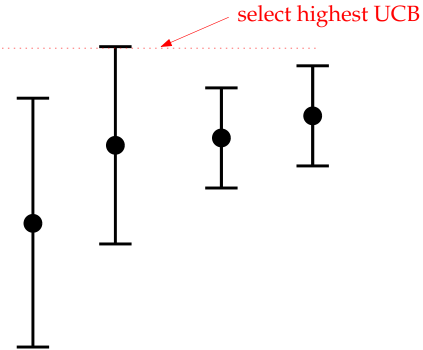


- highest UCB

$$\bar{V}_j + \sqrt{\frac{2\ln n}{n_j}}$$

Criterion

(Browne 2012; Browne et al. 2012)

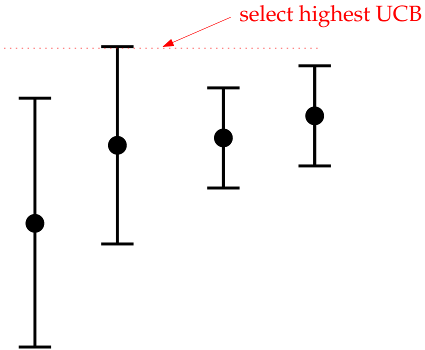


- highest UCB
- **not** highest reward

$$\bar{V}_j + \sqrt{\frac{2 \ln n}{n_j}}$$

Criterion

(Browne 2012; Browne et al. 2012)



- highest UCB
- **not** highest reward
- **not** widest spread

$$\bar{V}_j + \sqrt{\frac{2\ln n}{n_j}}$$

UCT Pseudocode

(Browne 2012)

The below was taken directly from Cameron Browne slides.

Algorithm 2: The UCT algorithm

function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD($v_0, 0$))
function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v
function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'
function BESTCHILD(v, c)
  return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$
function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s
function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

Theory and Practice of Artificial Intelligence

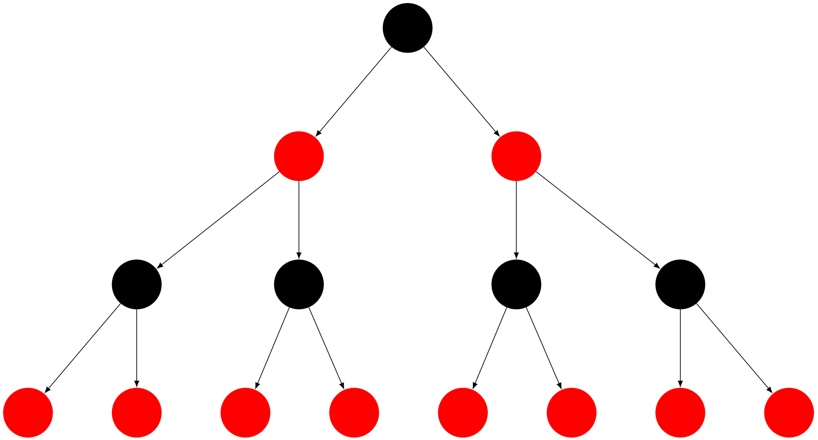
Other Game Types

Daniel Polani

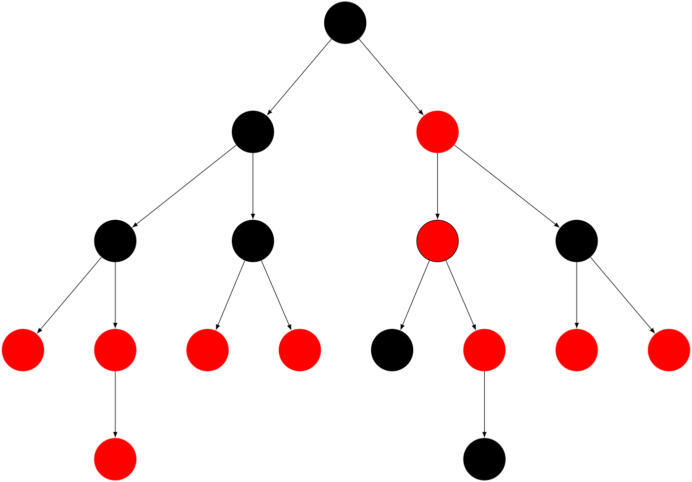
School of Computer Science
University of Hertfordshire

March 9, 2017

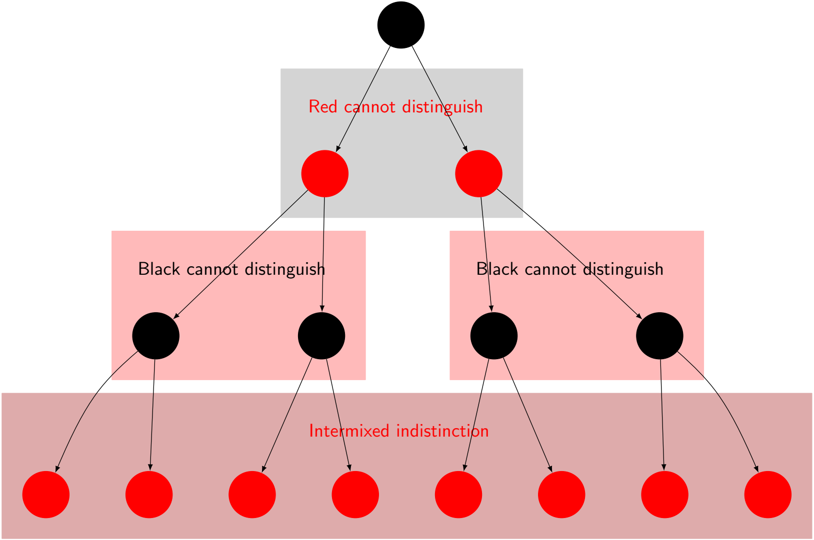
Game Tree I — Red and Black Alternate

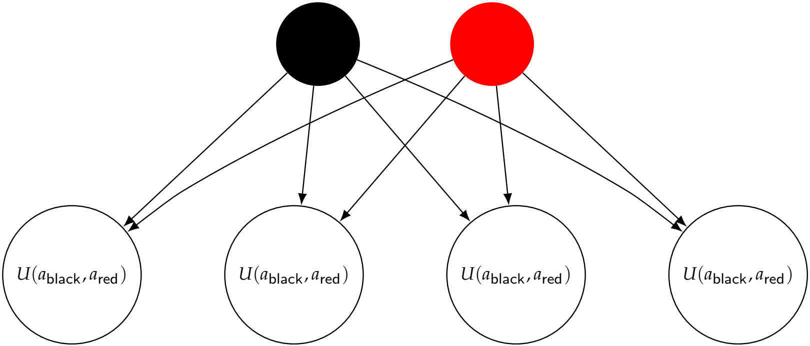


Game Tree II —
Red and Black Alternate Irregularly



Game Tree III — Hidden Info




$U(a_{\text{black}}, a_{\text{red}})$
$U(a_{\text{black}}, a_{\text{red}})$
$U(a_{\text{black}}, a_{\text{red}})$
$U(a_{\text{black}}, a_{\text{red}})$

Other Game Types

Prisoner's Dilemma: game with

1. simultaneous moves
2. non-zero-sum payoff

$_{P1} \backslash ^{P2}$	defect	cooperate
defect	$_{-6} \backslash ^{-6}$	$_{0} \backslash ^{-10}$
cooperate	$_{-10} \backslash ^{0}$	$_{-1} \backslash ^{-1}$

Dominance

Def. (strong dominance): a strategy s for a player p *strongly dominates* s' if the payoff using s is better than using s' for every *fixed* choice of strategy for other players.

Def. (weak dominance): a strategy *weakly dominates* if it is better on (at least) one strategy of other players and no worse on any other.

Def.: A *dominant strategy* dominates all others.

Pareto optimality/dominance

Def. (Pareto optimality): an outcome is *Pareto optimal* if no other outcome would be preferred by *all* the players.

Def. (Pareto dominance): an outcome is strongly *Pareto dominated* if all players would prefer some other outcome

Def. (weak Pareto dominance): an outcome is weakly Pareto dominated, if some players would prefer another outcome to which all others would not mind switching

Dominance in Prisoner's Dilemma

Note: both Alice and Bob have a dominant strategy, i.e. we have a dominant strategy *equilibrium*

Def. (Nash equilibrium): a selection of strategies for each player such that no player can benefit by switching his/her strategy if all other players' strategies are unchanged.

Remark: the *dilemma* in the prisoner's dilemma is due to the fact that the Nash equilibrium $(-6, -6)$ of both prisoners defecting is Pareto dominated by $(-1, -1)$ of both prisoners cooperating.

Note: a Nash equilibrium can arise even without the existence of a dominant strategy.

Comments on Game Theory

Remark: if

- the prisoner's dilemma game is being iterated
- the players are allowed to have memories and identify their opponent

this can lead to solutions which avoid the equilibrium.

Note: Tit-For-Tat and very related strategies prove to be remarkably stable and robust solutions.

Remark: if one has a Pareto-optimal point which is also a Nash equilibrium, then we call that a *solution* of the game.

Back to Zero-Sum Games

Consider: simultaneous zero-sum games. Need to consider only the payoff P for one of the players, the other will follow as $-P$.

2-Finger Morra: payoff matrix:

$E \backslash O$	1	2
1	$2 \backslash -2$	$-3 \backslash 3$
2	$-3 \backslash 3$	$4 \backslash -4$

Goal: find *solution*

Zero-Sum Games: Solution

Scenario 1: force E to begin, O to follow. This is an advantage for O. Thus, E is guaranteed an outcome of $U_E \geq -3$.

Scenario 2: force O to begin, E to follow. O can ensure an outcome with $U_E \leq 2$.

Mixed Strategy

Note: revealing a strategy gives the second player an advantage.
For, if second player plays $[p:1;(1-p):2]$ (notation: lottery where outcome 1 is selected with probability p and outcome 2 is selected with probability $1-p$), the expected utility for E is

$$pU_E(O=1)+(1-p)U_E(O=2)$$

If $U_E(O=1)$ and $U_E(O=2)$ are different, O should pick the best as *pure* strategy.

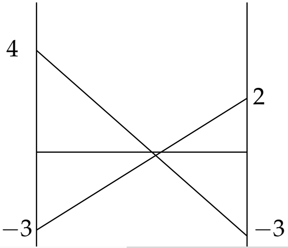
Utilities for Mixed Strategies I

Assume: E moves first, O does not know the move, but knows p in E's strategy $[p:1;(1-p):2]$. Then, if

1. O chooses 1, then
   $\mathbf{E}(U) = 2p - 3(1-p) = 5p - 3$
2. O chooses 2, then
   $\mathbf{E}(U) = -3p + 4(1-p) = 4 - 7p$.

Thus:

- O will always pick the minimum of both
- E will pick p such that this minimum is maximal
- i.e. resulting payoff is $U = -\frac{1}{12}$.



Utilities for Mixed Strategies II

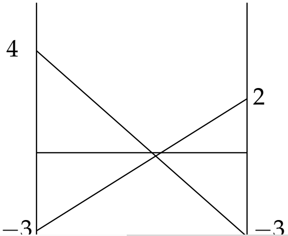
Assume: O moves first, probabilites $[q : 1; (1-q) : 2]$. If

1. E picks 1, then $\mathbf{E}(U) = 2q - 3(1-q) = 5q - 3$
2. E picks 2, then $\mathbf{E}(U) = -3q + 4(1-q) = 4 - 7q$

Thus:

- E picks the maximum of both
- O picks q such that this maximum is minimal
- i.e. value becomes $U = -\frac{1}{12}$.

Note: The two U values enclose the true value, which is therefore $U = -\frac{1}{12}$. It turns out that $p = \frac{7}{12} = q$.



Minimax Equilibria

Bottom Line: there exists an *equilibrium*, a *minimax* equilibrium which is Nash equilibrium.

von Neumann: every two-player zero-sum game has a minimax equilibrium on mixed strategies. Also, in zero-sum games, Nash equilibria are minimax equilibria.

Theory and Practice of Artificial Intelligence

Dynamic Programming and MDPs

Daniel Polani

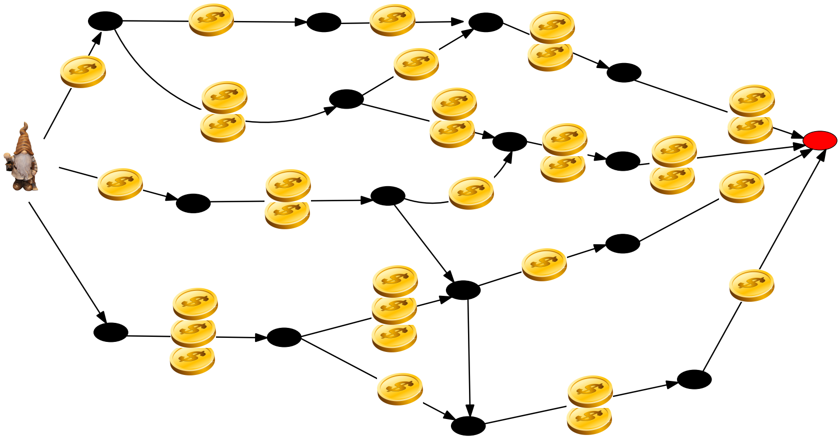
School of Computer Science
University of Hertfordshire

March 9, 2017

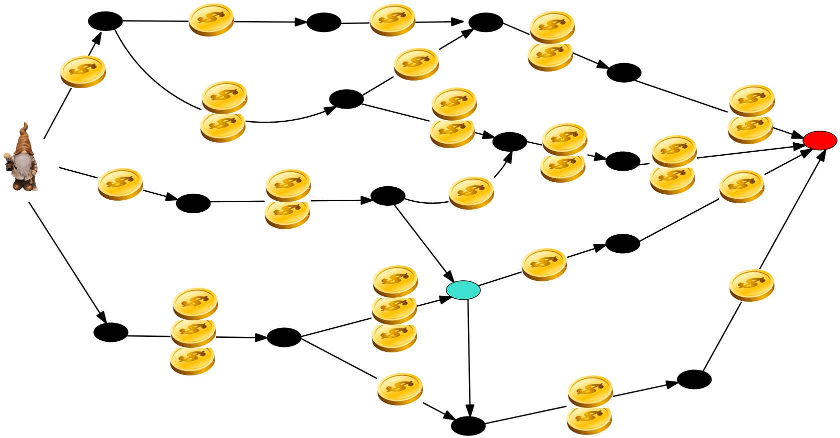
MDP: Motivation

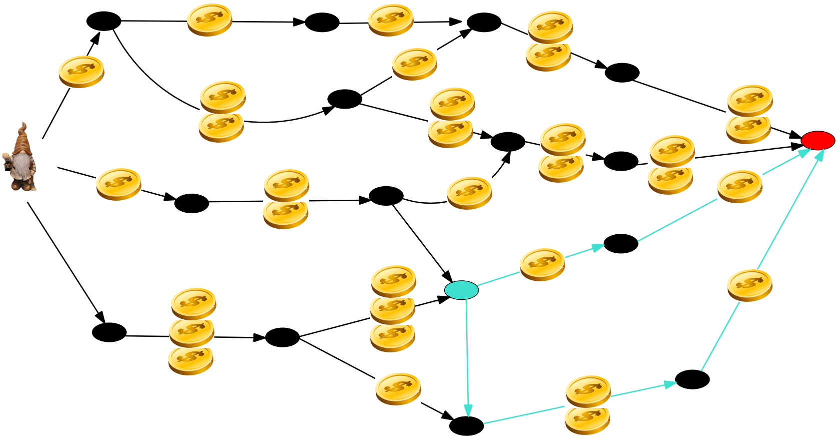
Scenario: sequence of decisions where

1. each decision may lead randomly to different outcomes
2. each decision is connected with a reward
3. rewards cumulate to total utility
4. rewards may be delayed

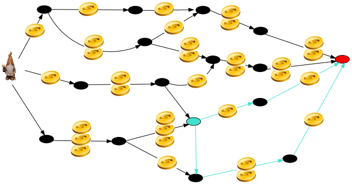


Scenario: Collecting Rewards





Scenario: Collecting Rewards



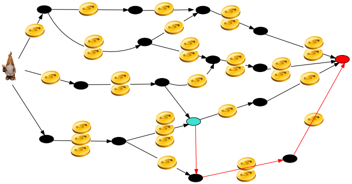
Task

- given a directed graph whose edges are labeled with rewards (here: coins)
- collect coins over paths
- choose path with most coins to collect

Task Structure

- reward on future paths from given vertex
- does **not** depend on past path

Scenario: Collecting Rewards



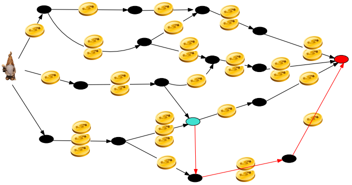
Task

- given a directed graph whose edges are labeled with rewards (here: coins)
- collect coins over paths
- choose path with most coins to collect

Task Structure

- reward on future paths from given vertex
- does **not** depend on past path

Scenario: Collecting Rewards



Task

- given a directed graph whose edges are labeled with rewards (here: coins)
- collect coins over paths
- choose path with most coins to collect

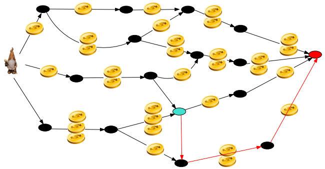
Task Structure

- reward on future paths from given vertex
- does **not** depend on past path

In Particular

Best path from current vertex v does not depend on the past!

Separability



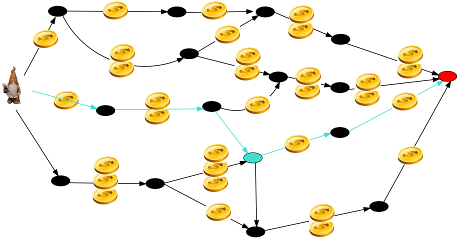
Motto

- whatever happened before does not affect future strategy
- whether good or bad collection happened before
- future collection should be optimal
- to optimize the rest

Corollary

- If a path is optimal from the beginning to the end
- any path from some intermediate vertex to the end is also optimal

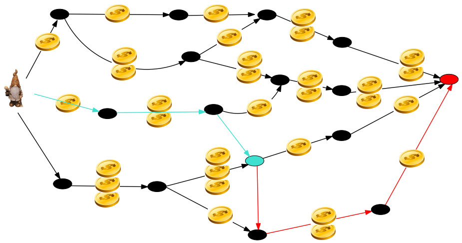
Separability



Proof Sketch

- Assume it were not so: i.e.
    - the total path $\langle v_0, v_1, \ldots, v_k \rangle$ is optimal
    - but the remaining subpath $\langle v_s, v_{s+1} \ldots v_k \rangle$ were not

Separability



Proof Sketch

- Assume it were not so: i.e.
  - the total path $\langle v_0, v_1, \ldots, v_k \rangle$ is optimal
  - but the remaining subpath $\langle v_s, v_{s+1} \ldots v_k \rangle$ were not
- Then, starting (unchanged) at v_s, we could improve the subpath to $\langle v_s, v'_{s+1} \ldots v'_k \rangle$.
- But then the complete path $\langle v_0, v_1, \ldots, v_s, v'_{s+1} \ldots v'_k \rangle$ would collect more reward than the original. $\lightning$

Bellmann Principle: Incredibly Powerful Idea!

1. cheapest paths
2. best paths
3. portfolio management
4. string comparison
5. genetic alignment
6. reinforcement learning (robotics and AI)
7. optimal control
8. Viterbi algorithms (e.g. in speech detection)
9. production optimization
10. and much more

The Bellmann Principle

Separability

- if, no matter what reward one collected
- one wishes the future reward to be maximal
- one has **separability**
- and **the Bellmann Principle** holds
- this is the case, e.g., if we simply add all rewards

Counterexamples: e.g. multiplication with vanishing or negative numbers

Then

Any suffix of an optimal path $\langle v_1, \ldots, \underbrace{v_l, \ldots, v_k}_{\text{suffix}} \rangle$ is also optimal.

The Bellmann Principle

Separability

- if, no matter what reward one collected
- one wishes the future reward to be maximal
- one has **separability**
- and **the Bellmann Principle** holds
- this is the case, e.g., if we simply add all rewards

Counterexamples: e.g. multiplication with vanishing or negative numbers

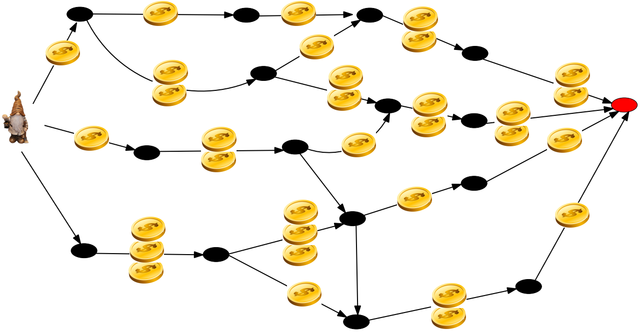
Then

Any suffix of an optimal path $\langle v_1, \ldots, \underbrace{v_l, \ldots, v_k}_{\text{suffix}} \rangle$ is also optimal.

Consequence

- Maximal future reward only depends on starting vertex v
- We say: $U^*(v)$ is the **optimal reward** starting at v
  The star * means **optimal**

In our example



Task

- find optimal path, i.e. collecting the most gold coins
- **Bellmann Principle:** every suffix of an optimal path is optimal itself
- this suggests a strategy

Strategy

Assumptions

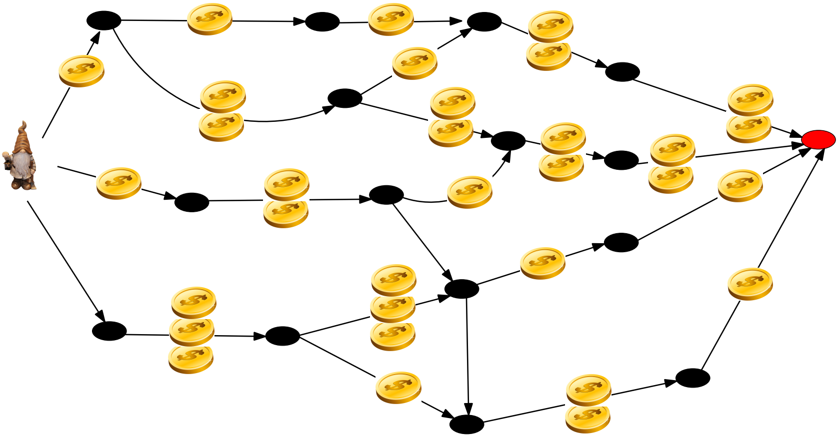
- **here**: we begin very simplest case only
- assume deterministic and no cycles, i.e. DAG with rewards

However, note: even cycles can sometimes be handled directly, as long as there are no arbitrage loops (Sedgewick and Wayne 2011), Chapter 4.4. We will see later how, in Reinforcement Learning, under use of further mathematical structure, even arbitrage loops can be treated!!

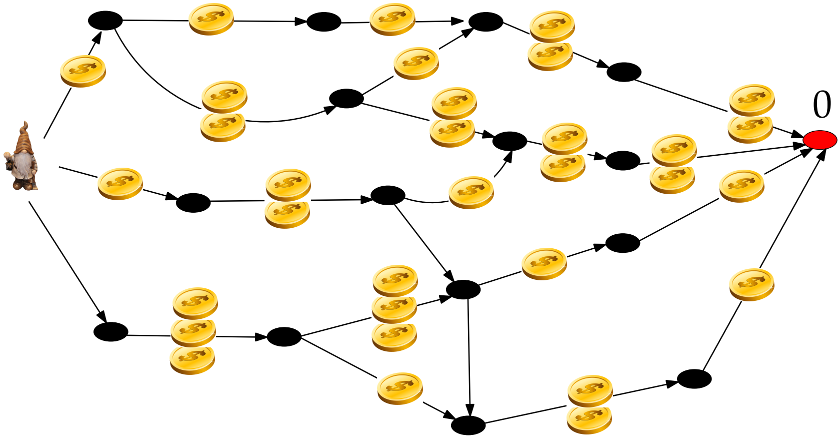
Dynamic Programming Algorithm

1. consider "end" vertices v (i.e. without outgoing edges)
   Exist, since we assumed a DAG
2. they won't collect any future reward, thus set $U^*(v) := 0$
3. for all vertices v where all successors v' (i.e. $(v, v') \in E$) have already computed $U^*(v')$:
   - compute $U(v) := \max_{v'} \left[r_{v \to v'} + U^*(v')\right]$
4. and repeat until no more such vertices exist
5. if vertices v remain with uncomputed $U^*(v)$, there is a loop
   why?

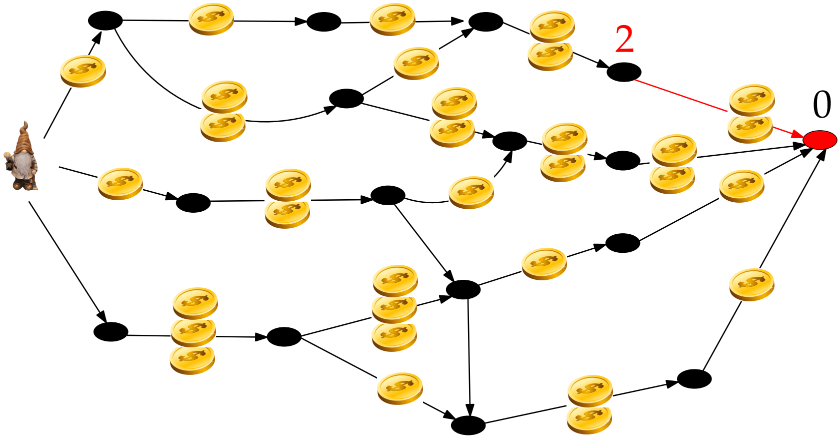
Rollback of Rewards



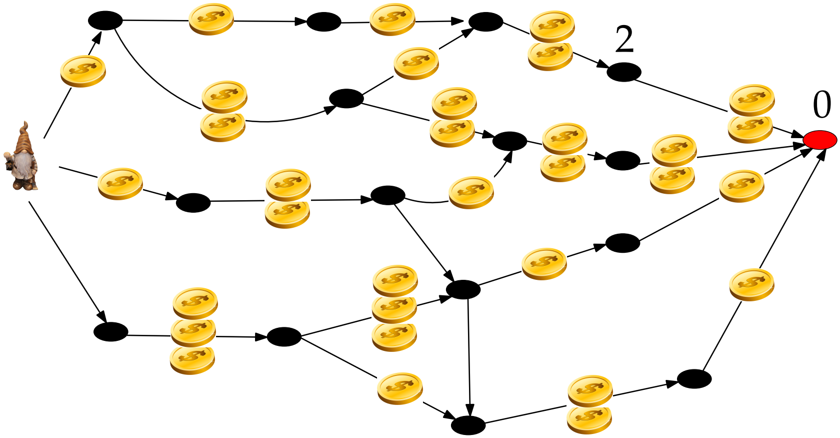
Rollback of Rewards



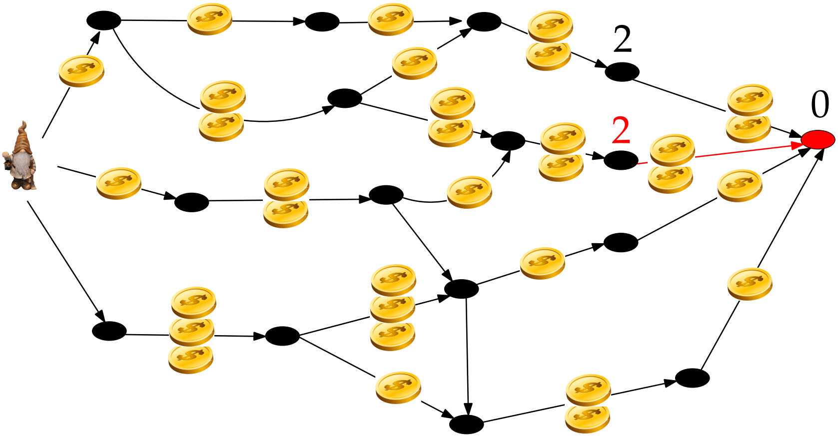
Rollback of Rewards



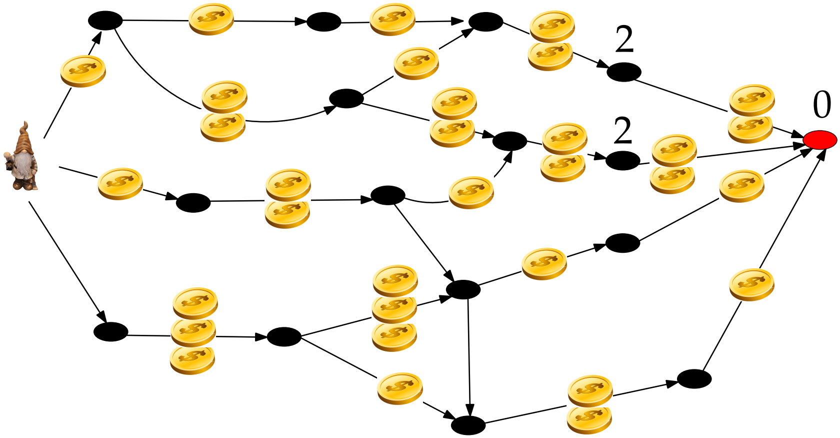
Rollback of Rewards



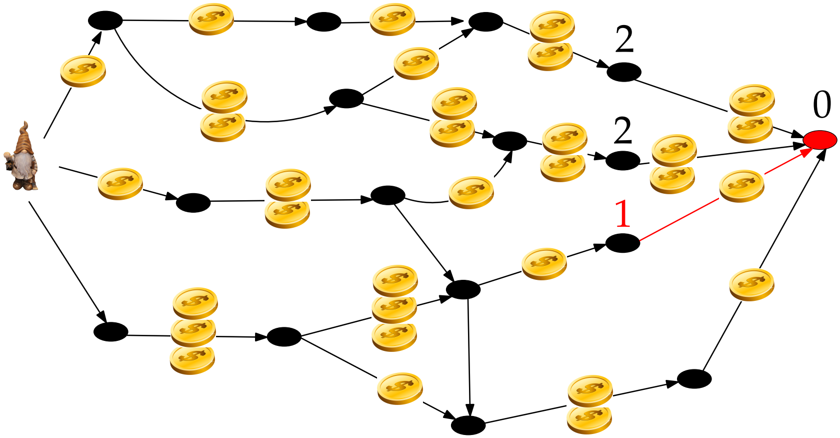
Rollback of Rewards



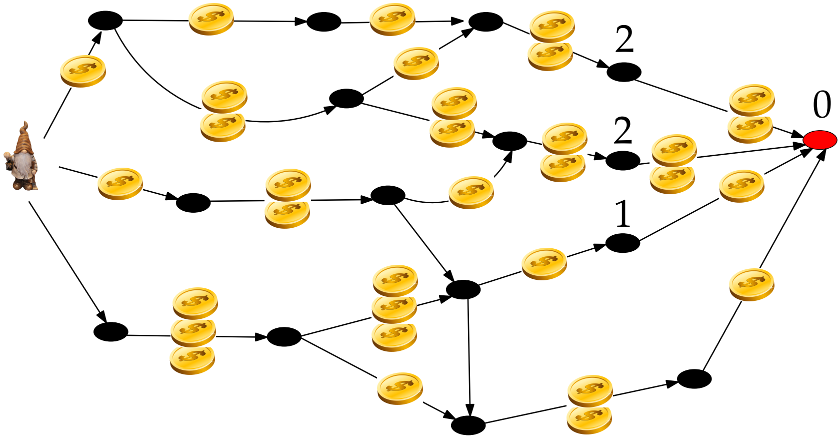
Rollback of Rewards



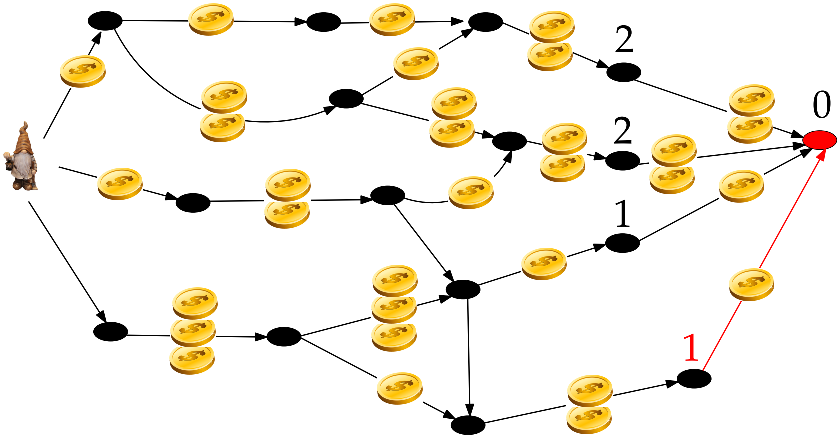
Rollback of Rewards



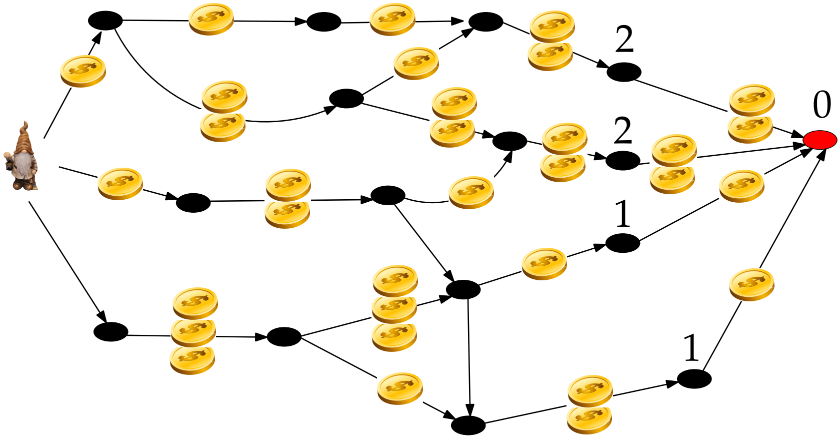
Rollback of Rewards



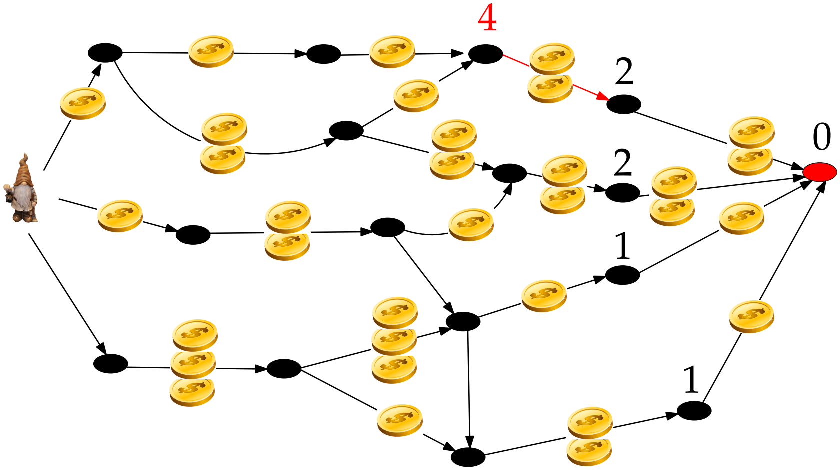
Rollback of Rewards



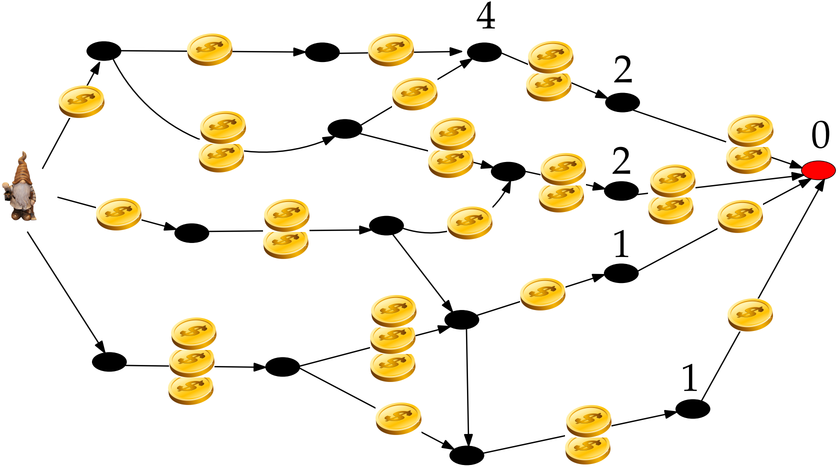
Rollback of Rewards



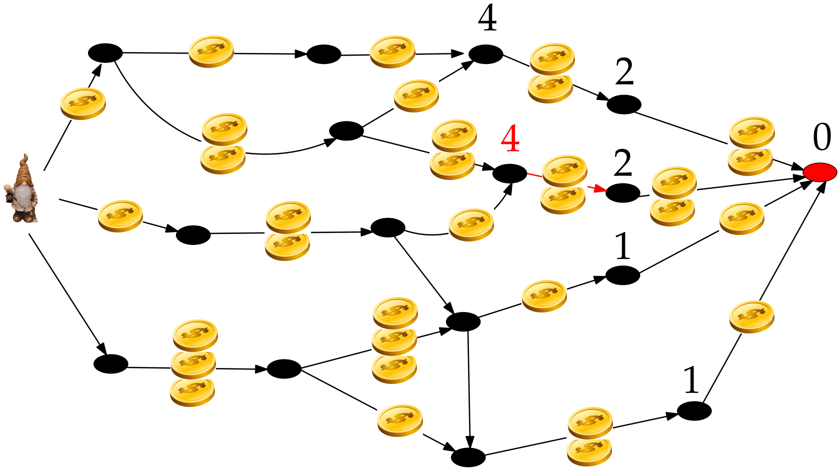
Rollback of Rewards



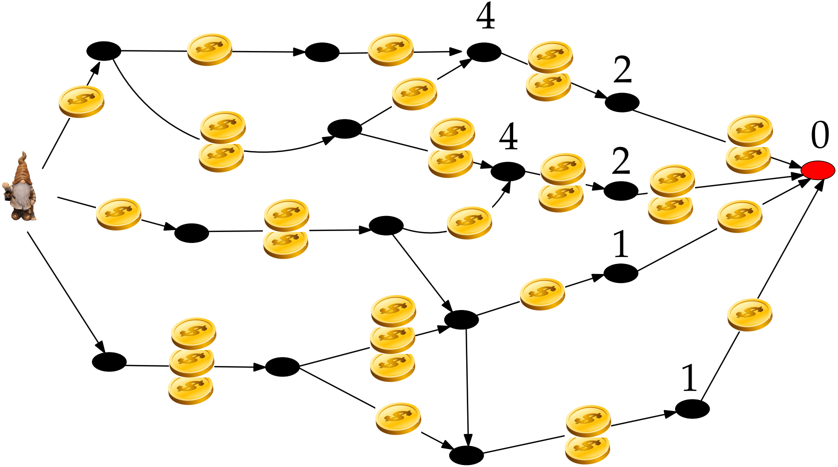
Rollback of Rewards



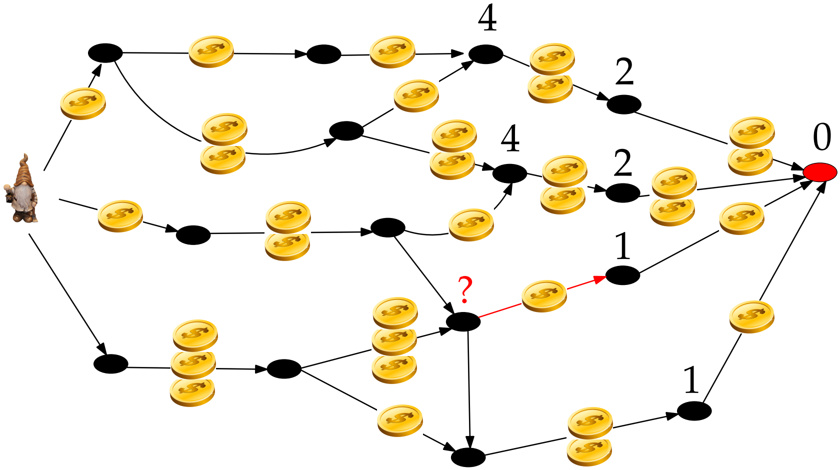
Rollback of Rewards



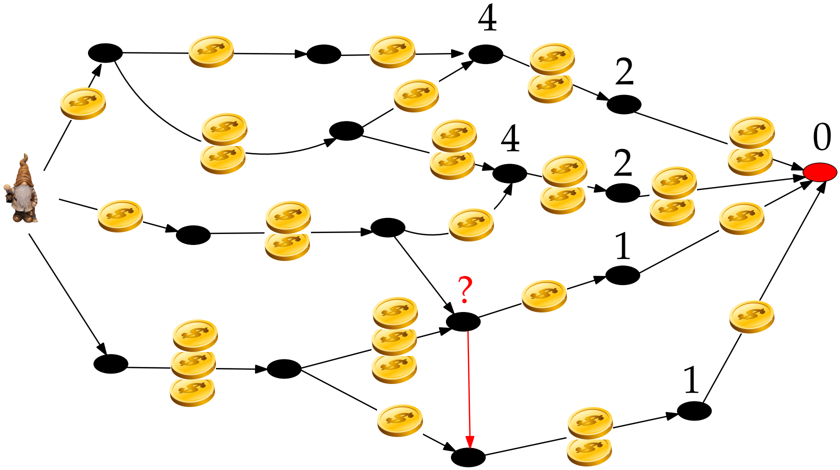
Rollback of Rewards



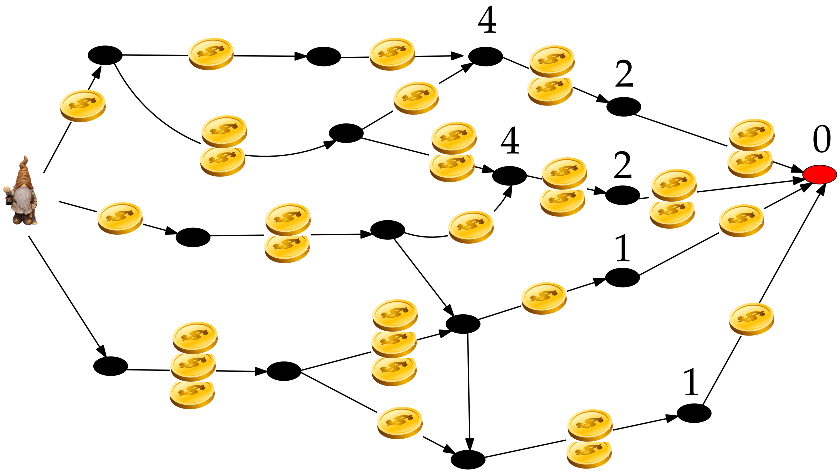
Rollback of Rewards

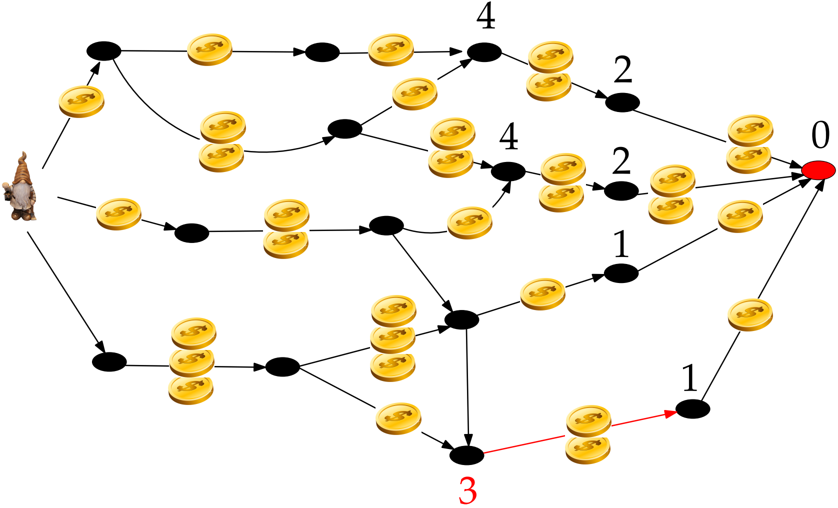


Rollback of Rewards

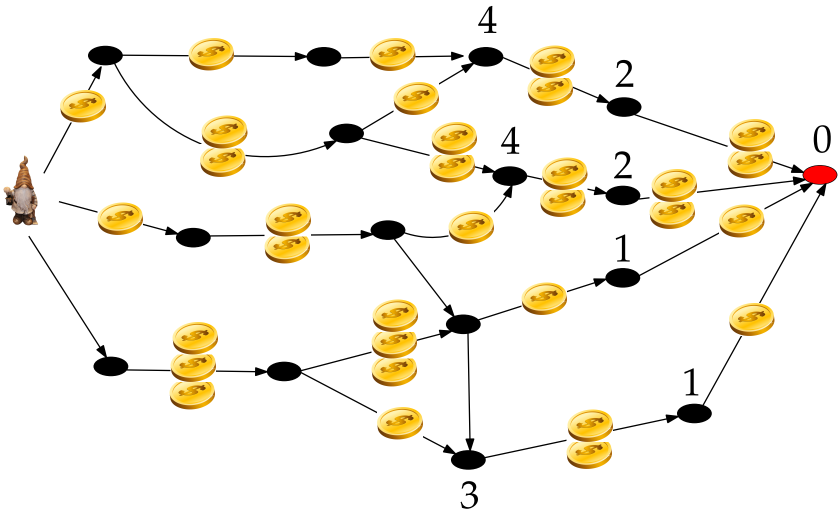


Rollback of Rewards

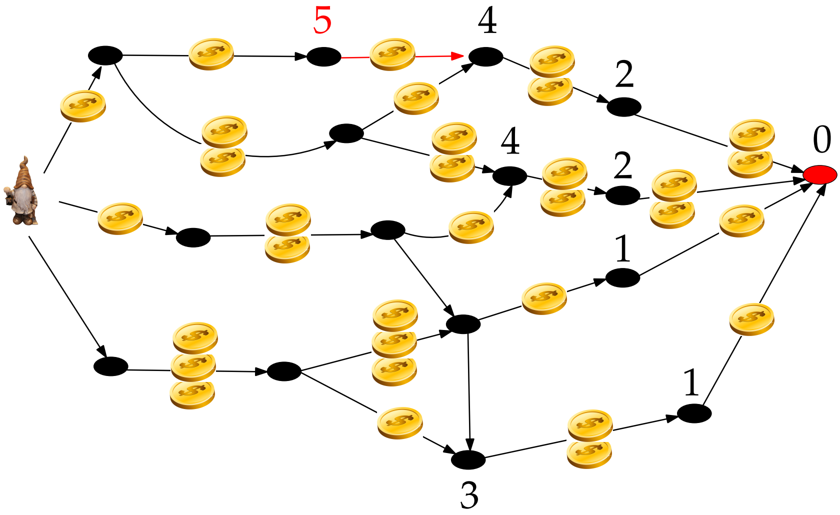



4
2
0
4
2
1
1
3

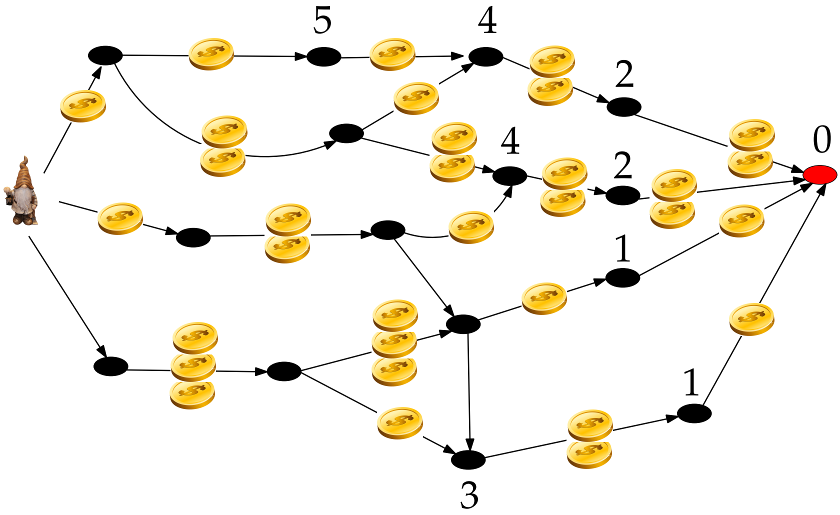
Rollback of Rewards



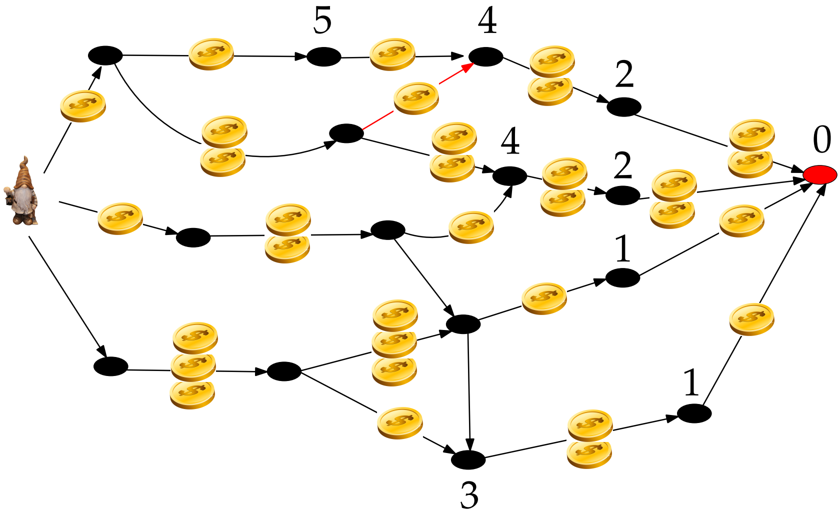
Rollback of Rewards



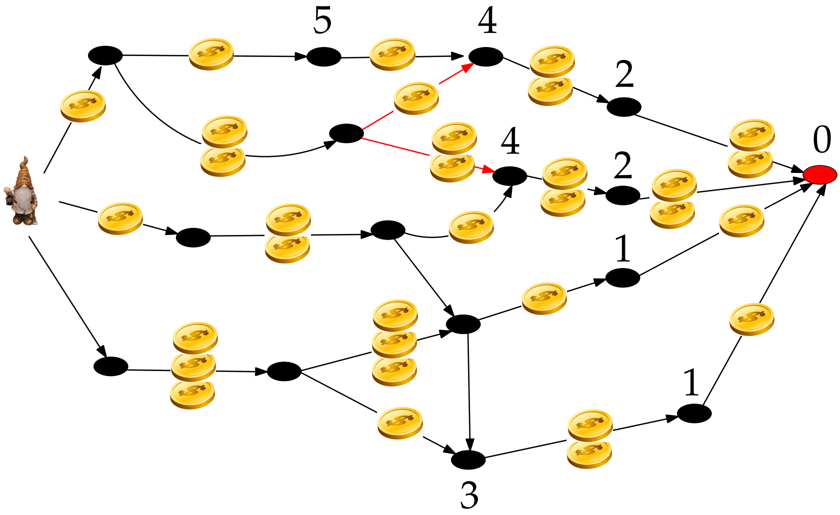
Rollback of Rewards



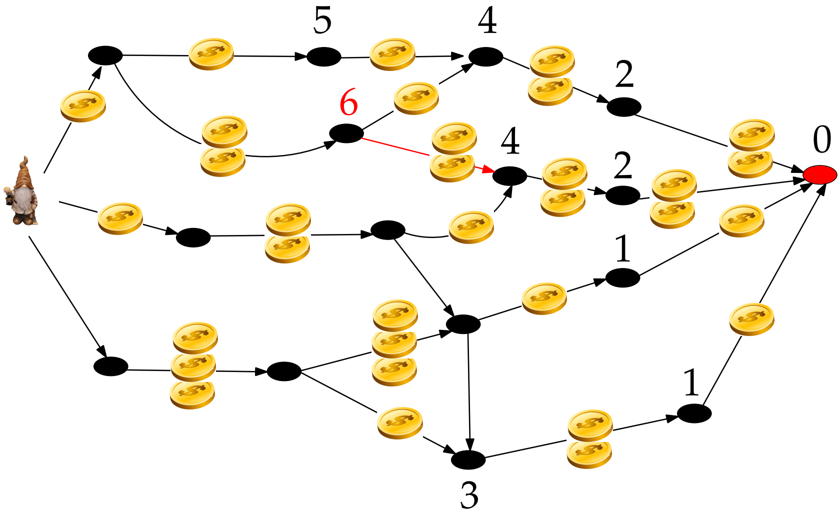
Rollback of Rewards



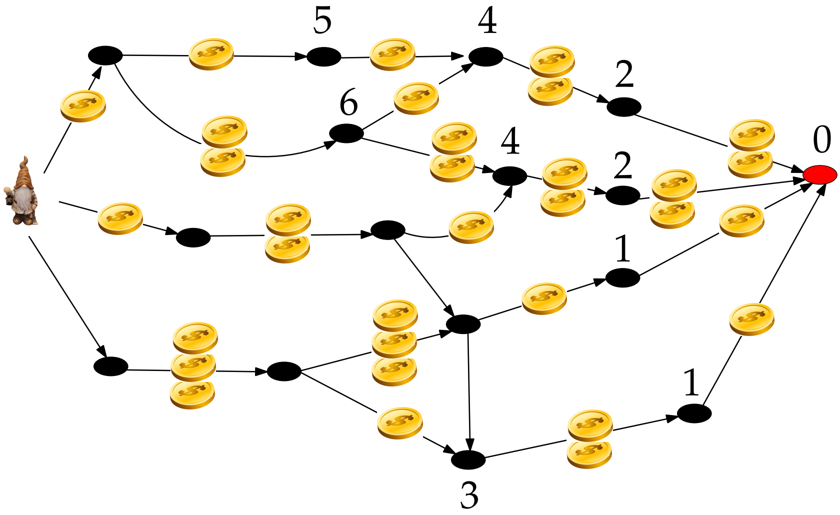
Rollback of Rewards

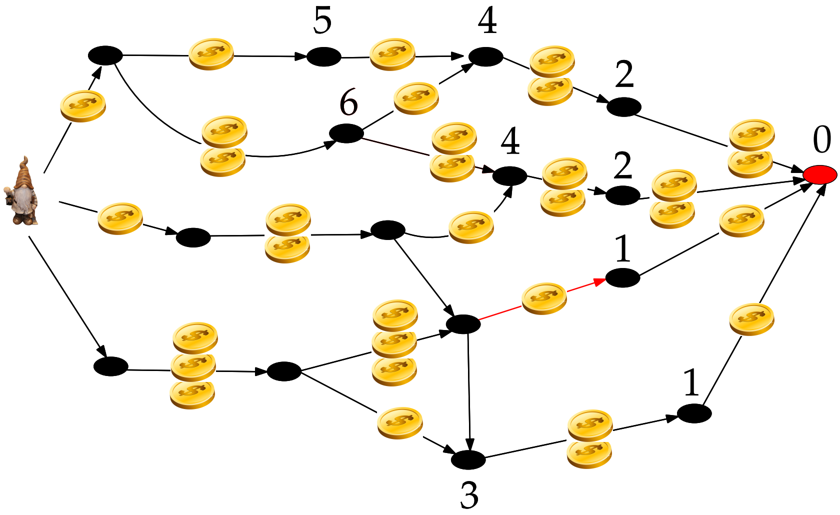


Rollback of Rewards

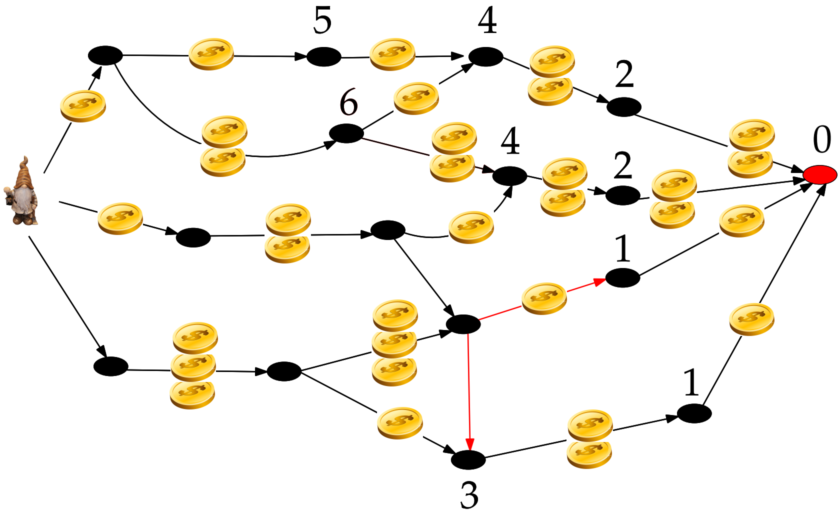


Rollback of Rewards

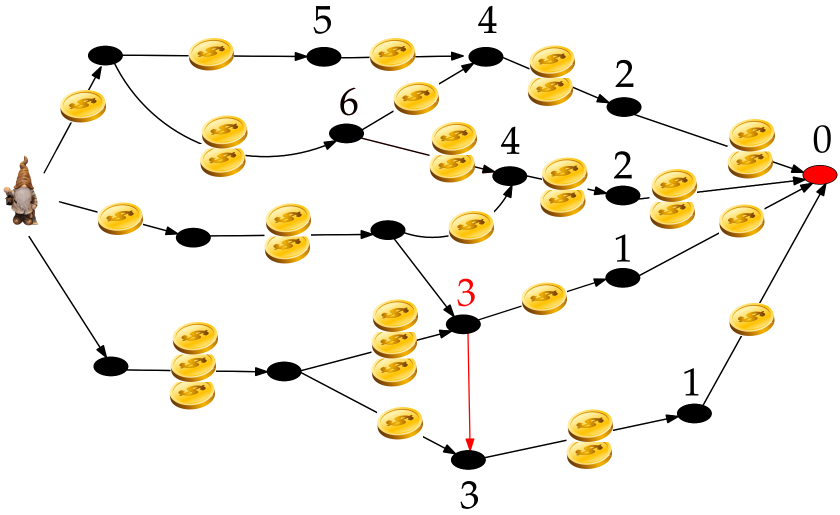



5
4
2
6
4
2
0
1
1
3

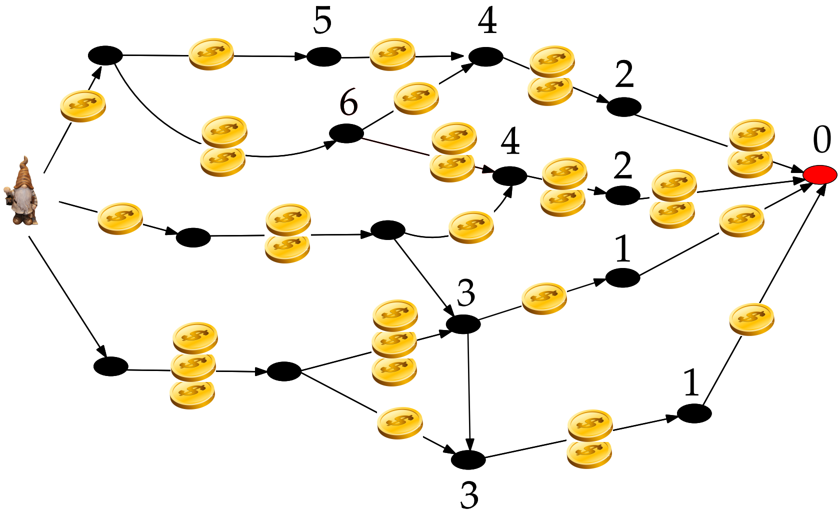
Rollback of Rewards



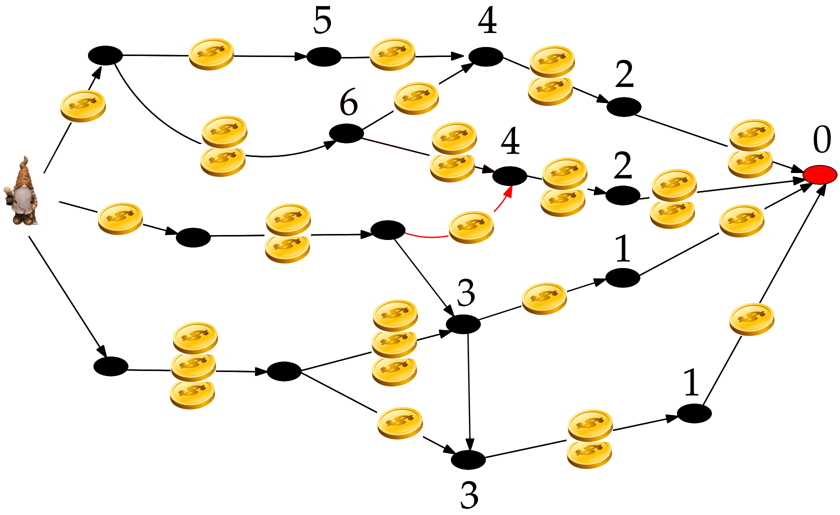
Rollback of Rewards

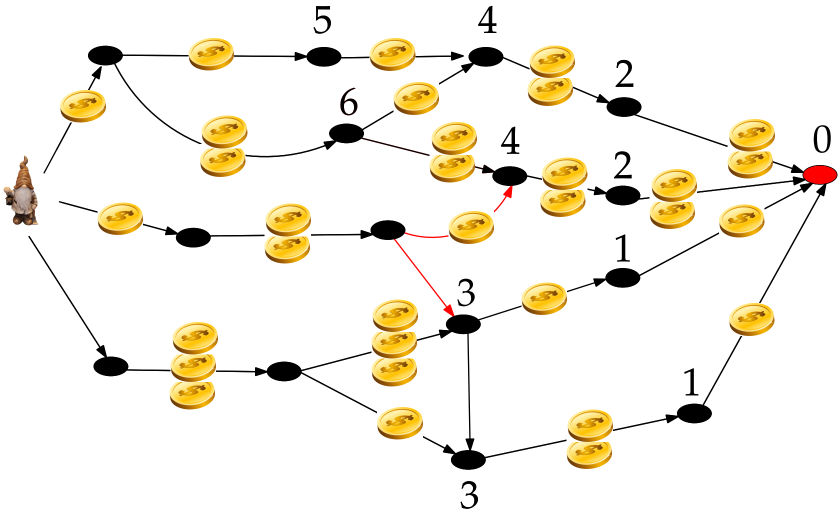


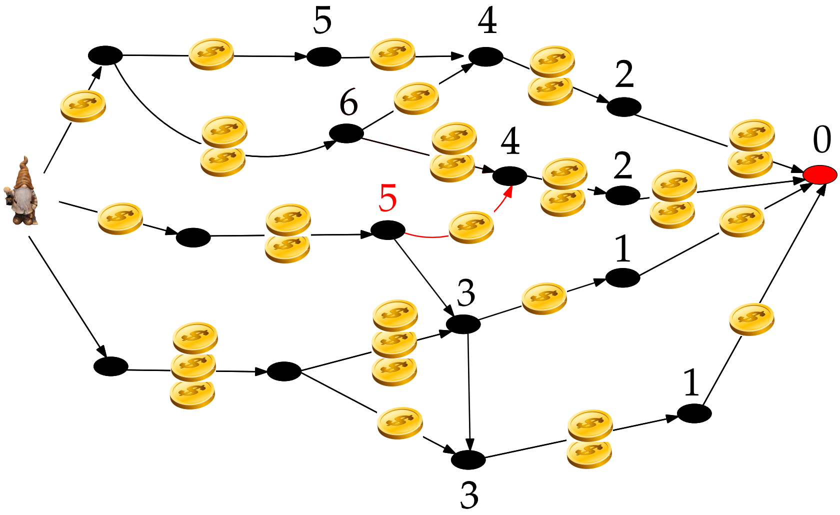
Rollback of Rewards



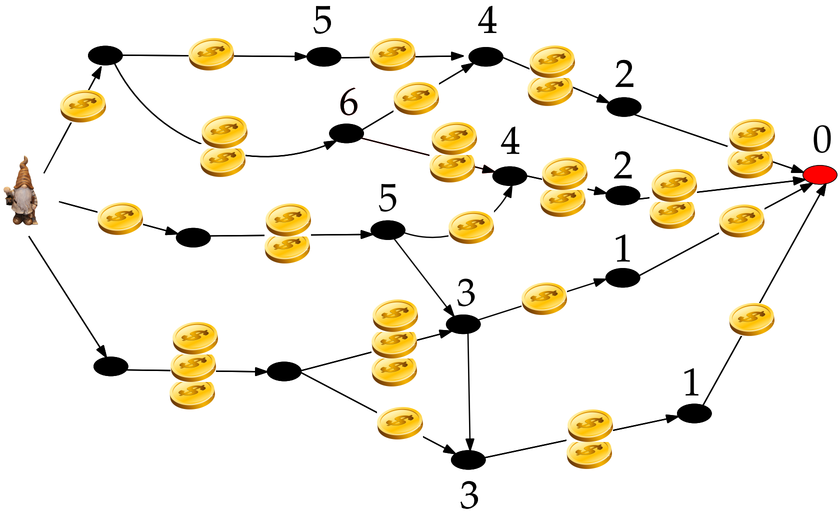
Rollback of Rewards



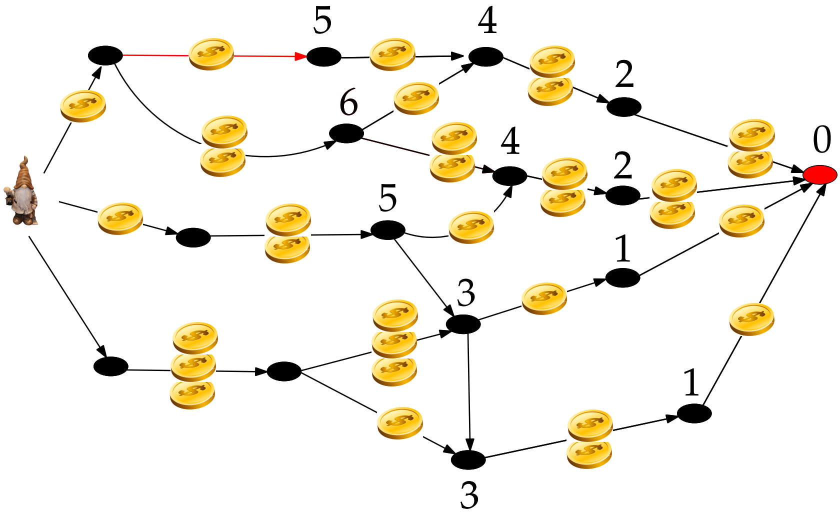

5
4
6
2
4
0
2
1
3
1
3


5
4
6
2
4
0
5
2
1
3
1
3

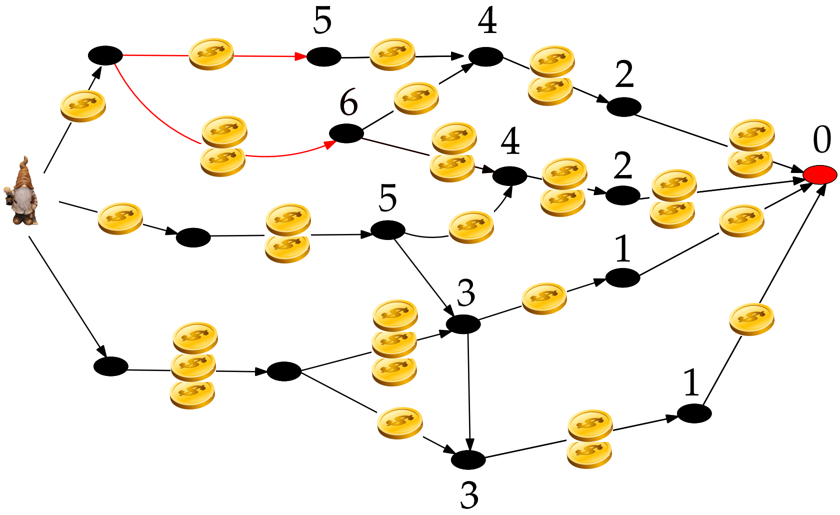
Rollback of Rewards



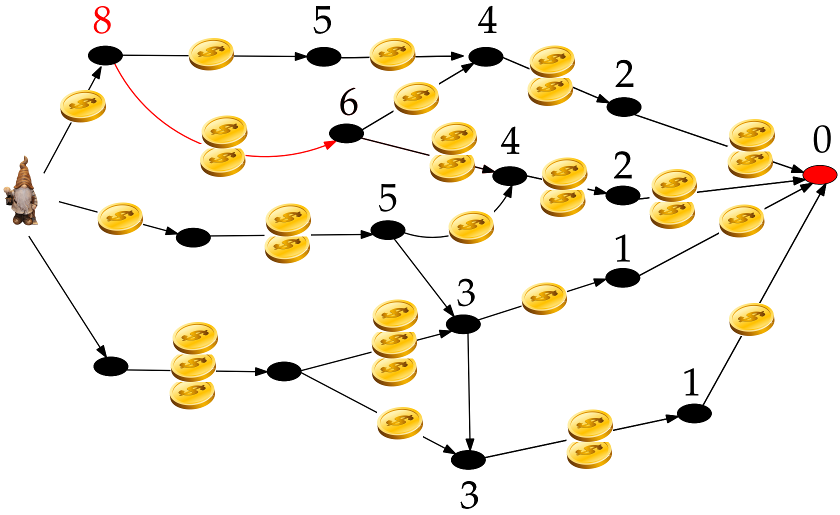
Rollback of Rewards



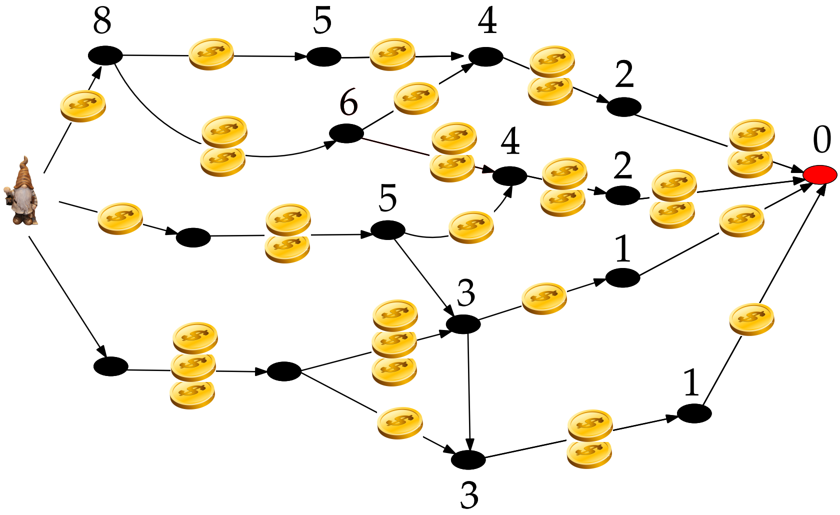
Rollback of Rewards



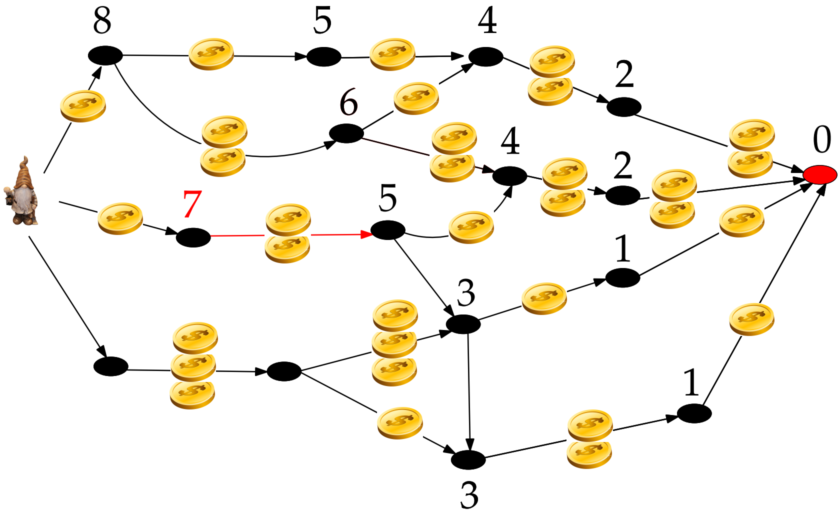
Rollback of Rewards



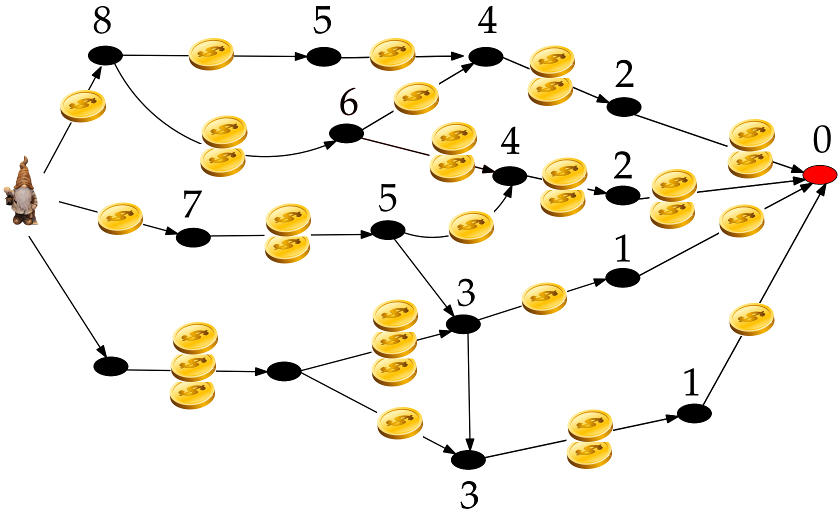
Rollback of Rewards



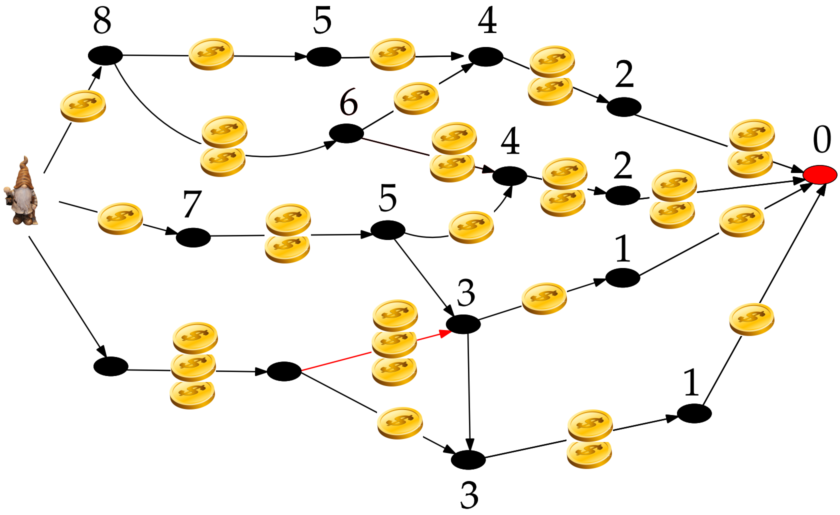
Rollback of Rewards



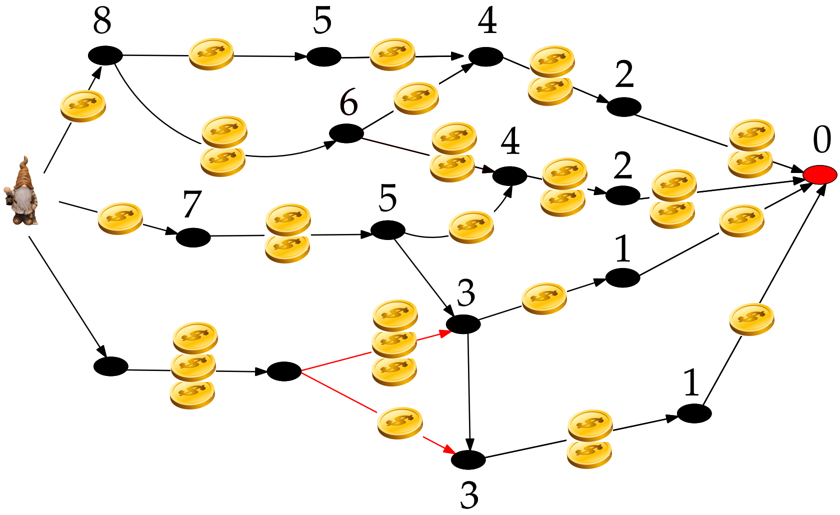
Rollback of Rewards



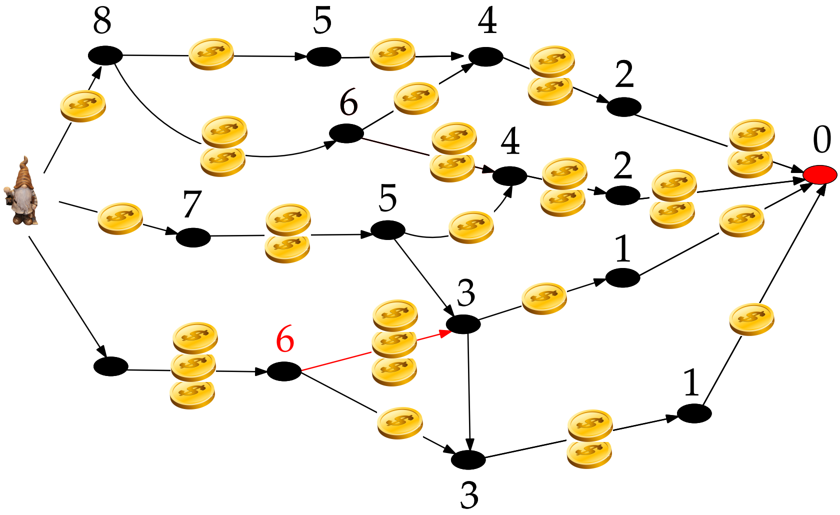
Rollback of Rewards



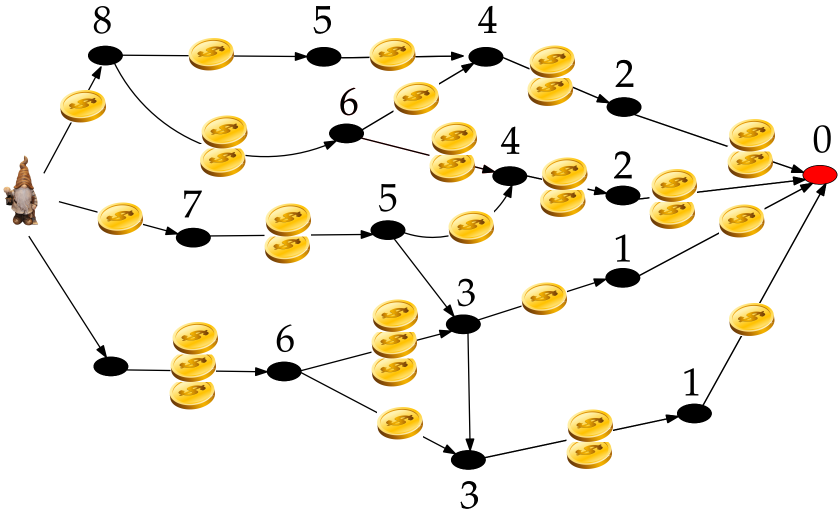
Rollback of Rewards



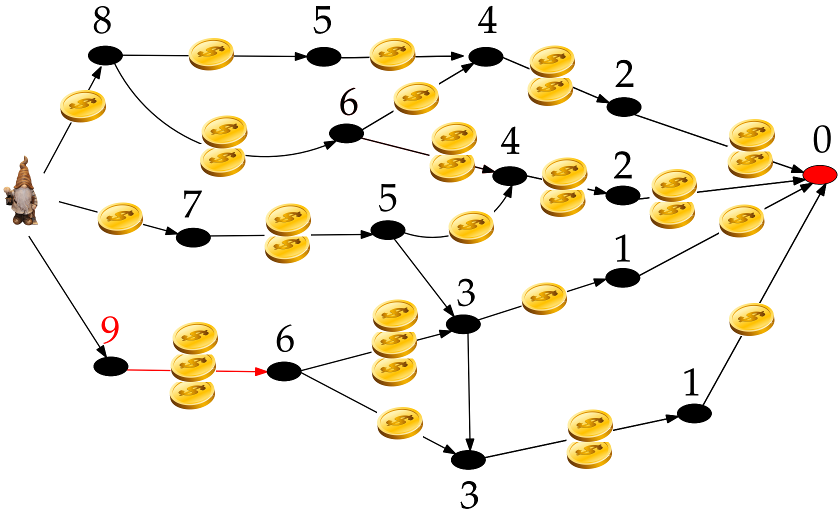
Rollback of Rewards



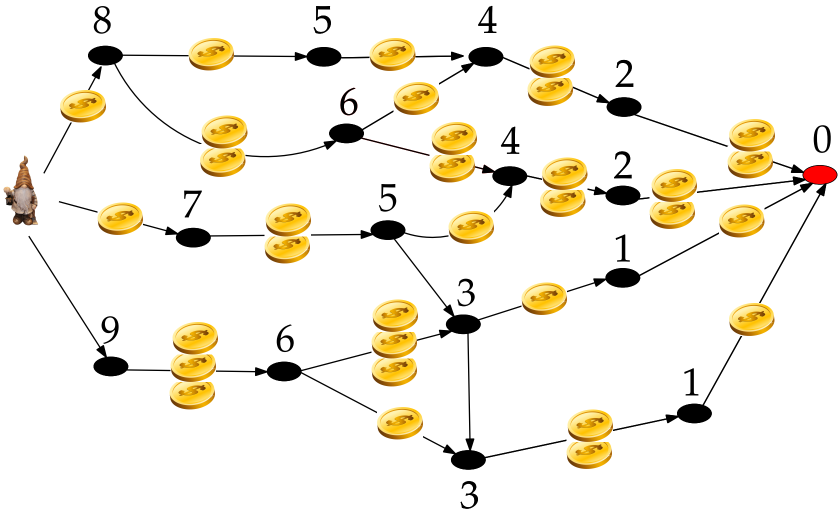
Rollback of Rewards



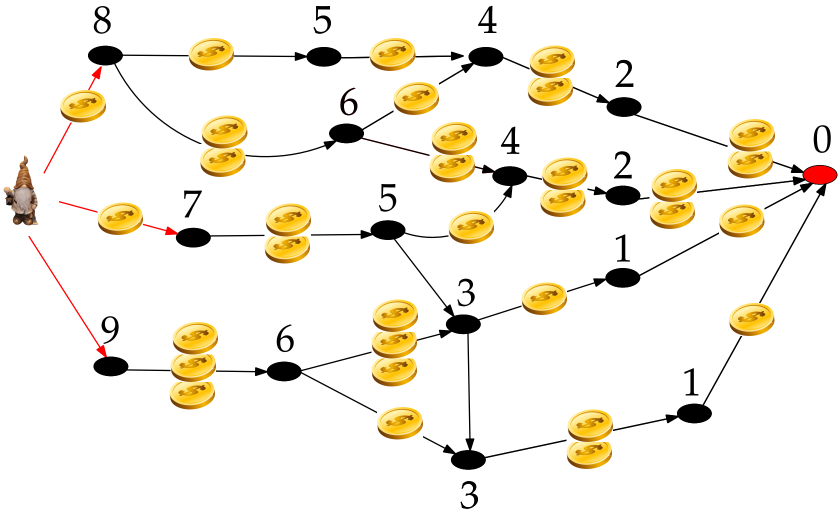
Rollback of Rewards

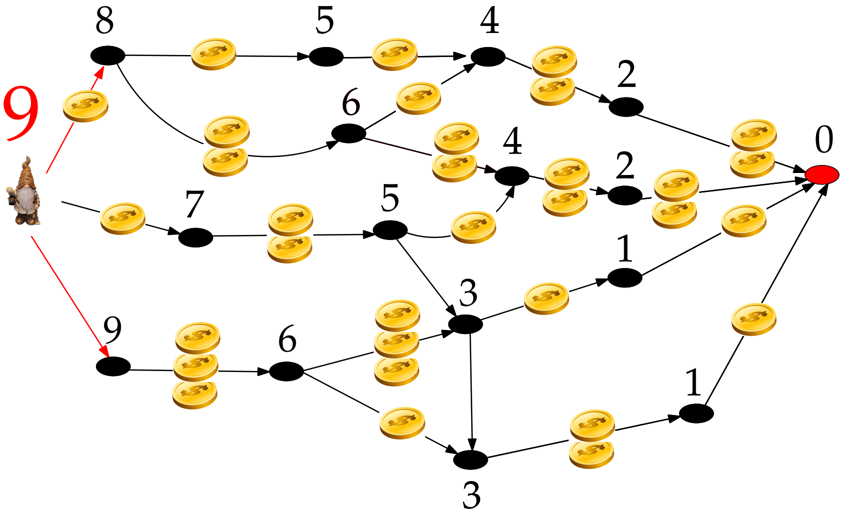


Rollback of Rewards

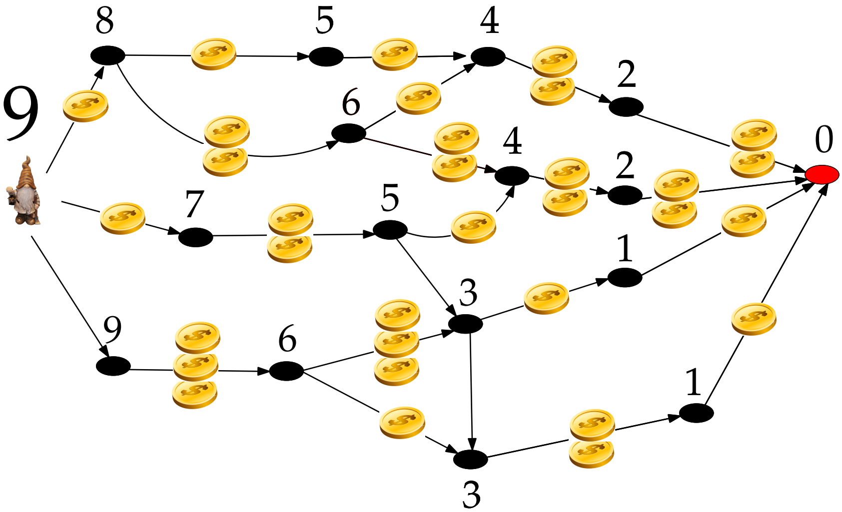


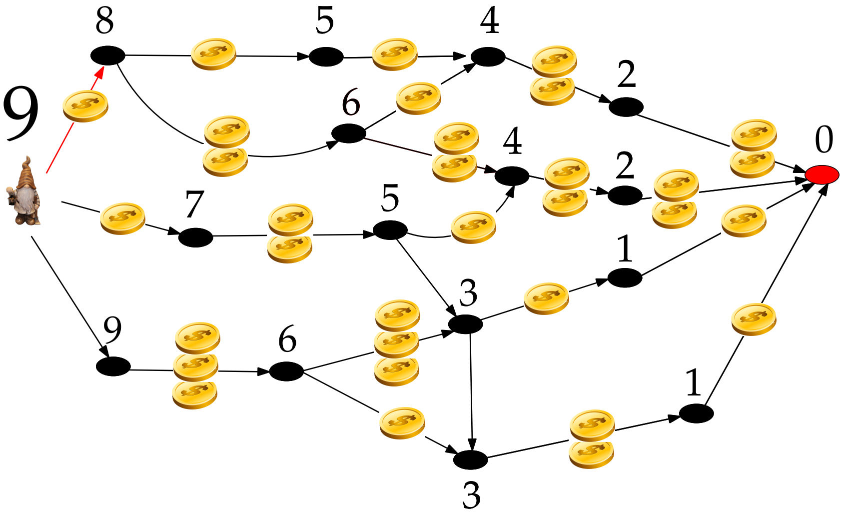
Rollback of Rewards

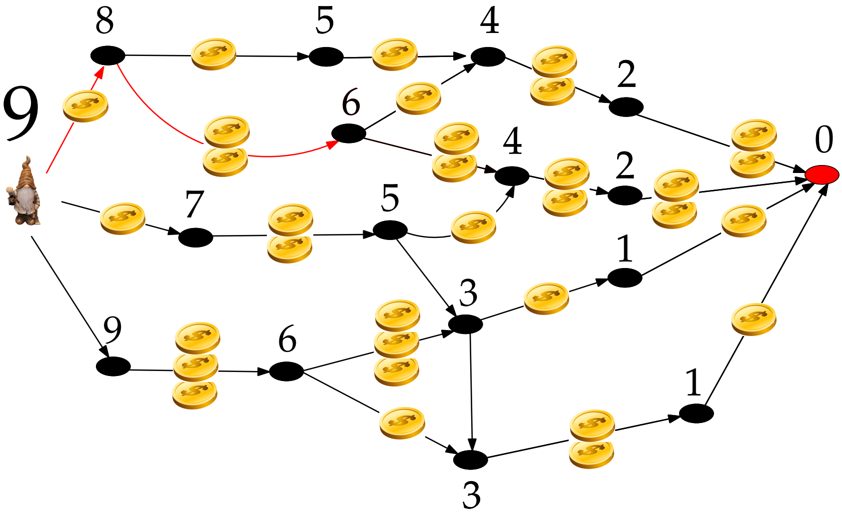



9
8
5
4
2
6
4
0
2
7
5
1
3
9
6
1
3

Rollback of Rewards

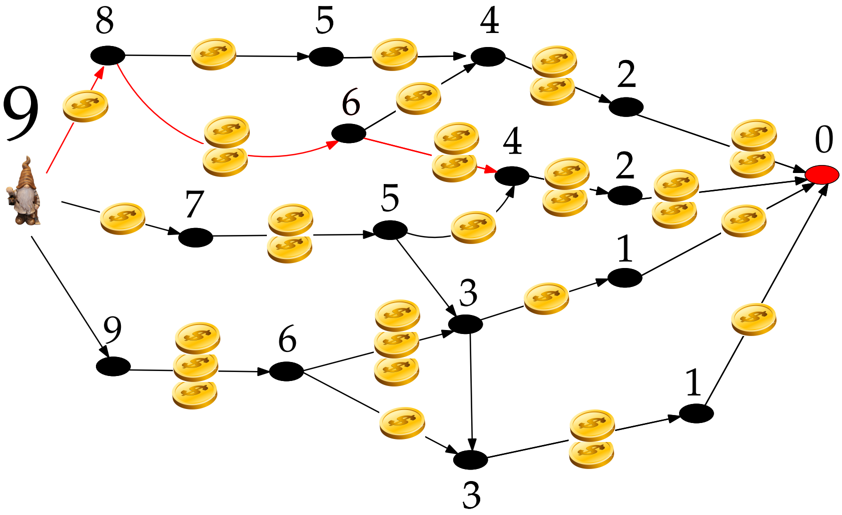



9
8
5
4
2
6
4
2
0
7
5
1
3
9
6
1
3

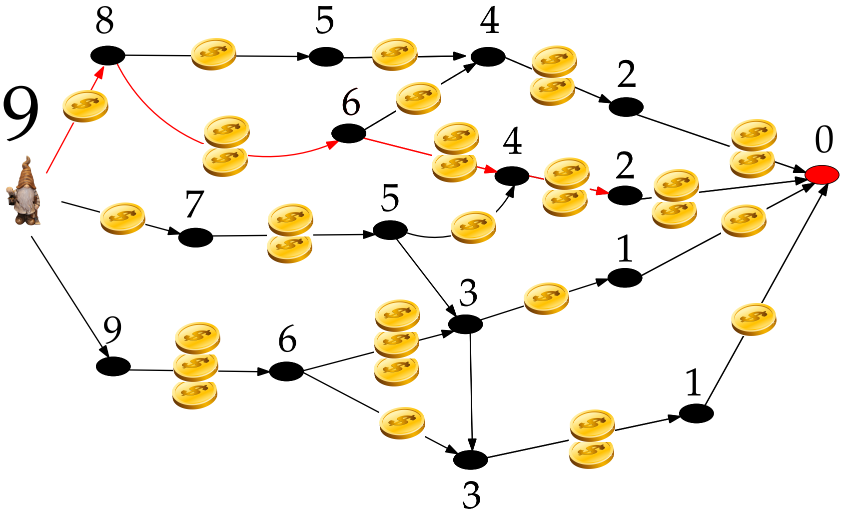


9
8
5
4
2
6
4
2
0
7
5
1
3
9
6
1
3

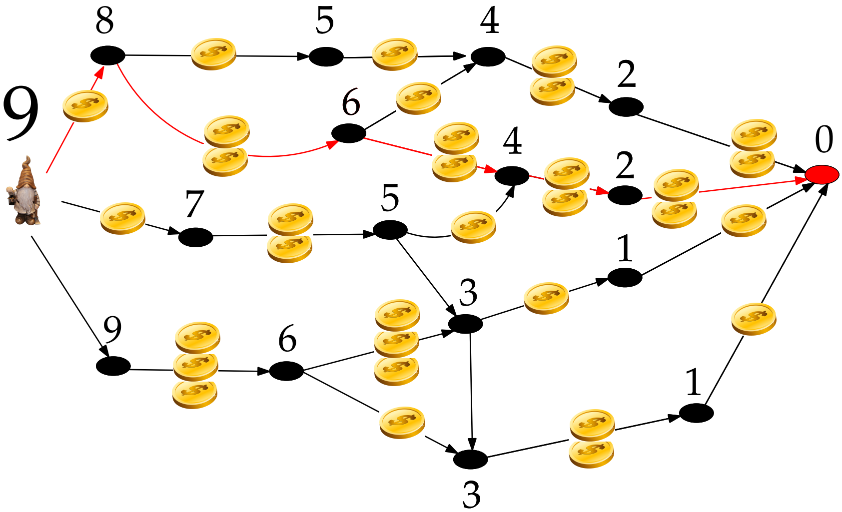
Rollback of Rewards



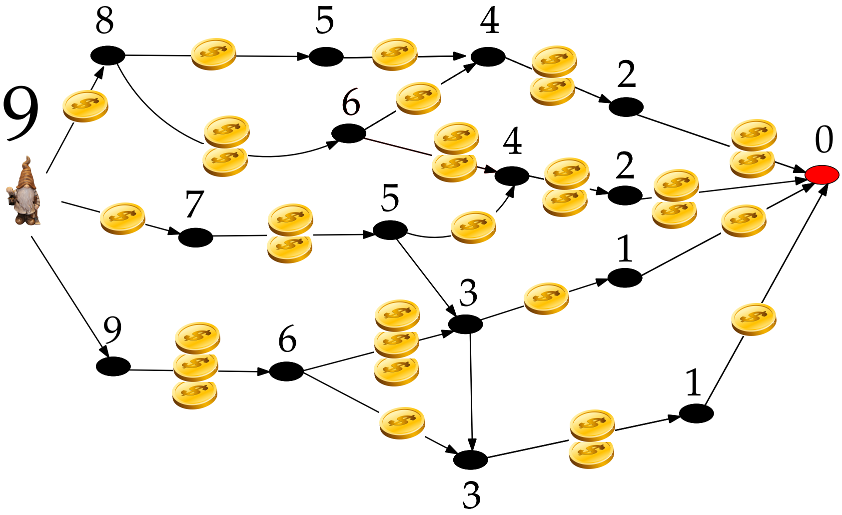
Rollback of Rewards



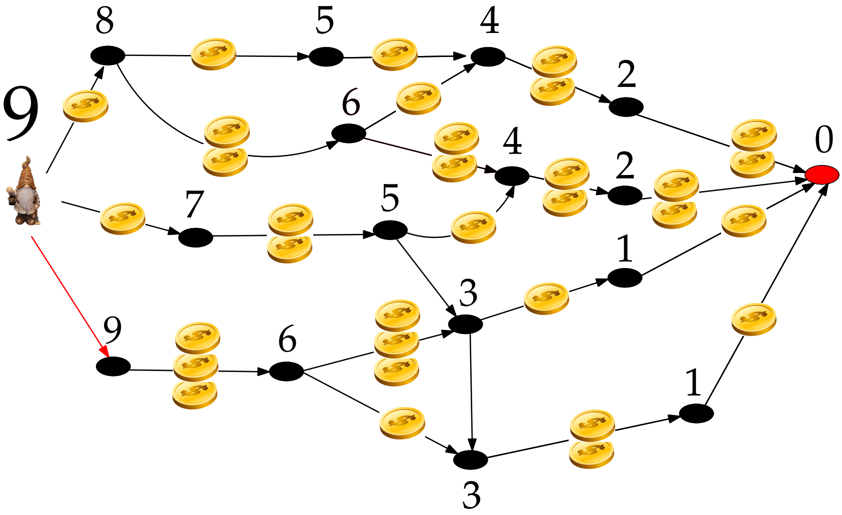
Rollback of Rewards



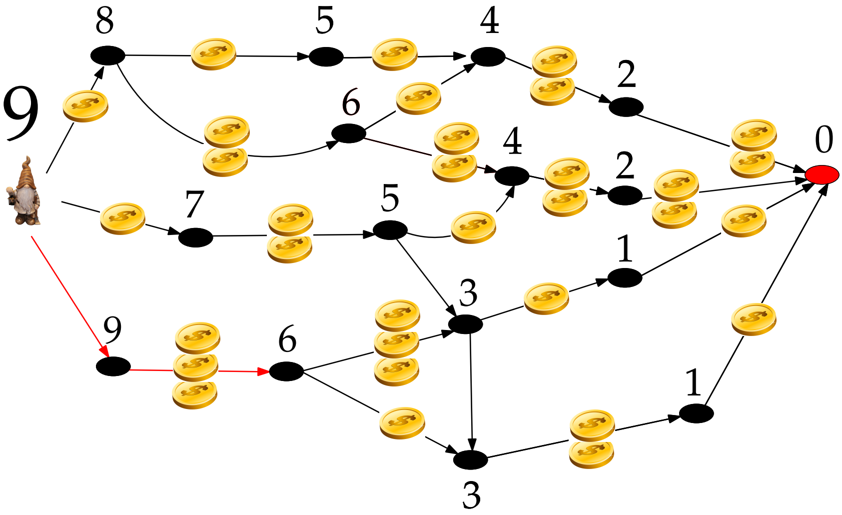
Rollback of Rewards



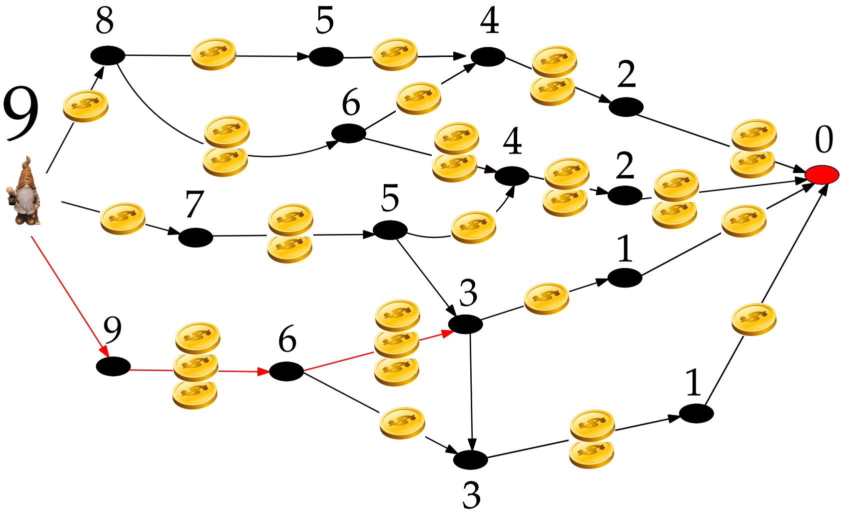
Rollback of Rewards



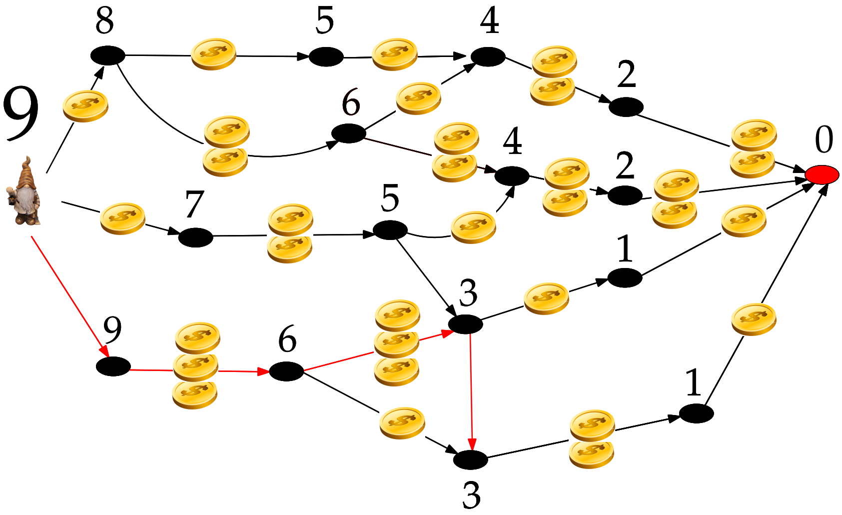
Rollback of Rewards



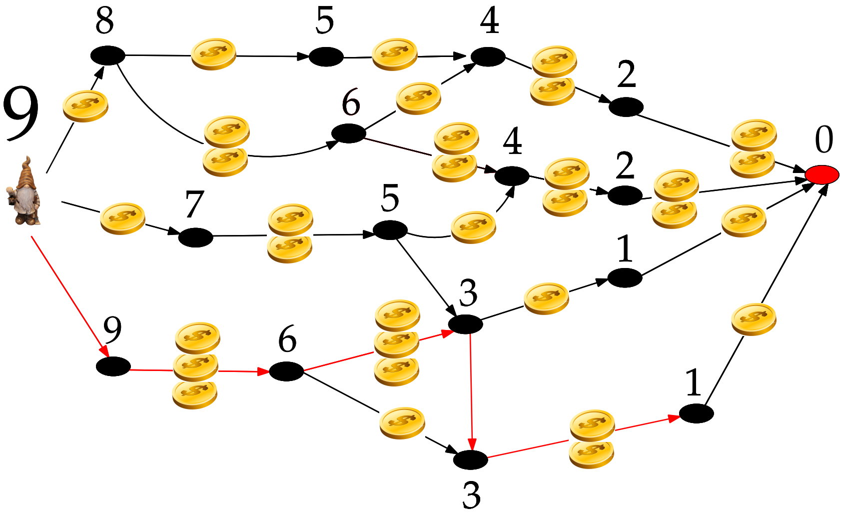
Rollback of Rewards

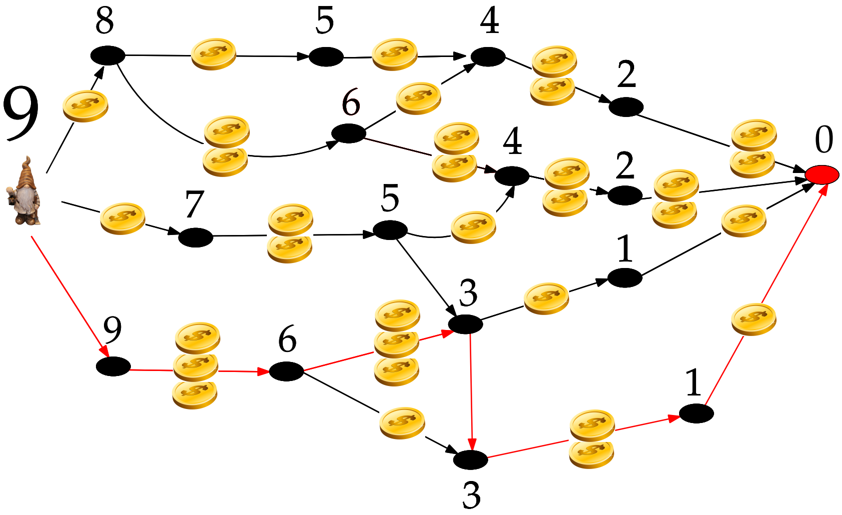


Rollback of Rewards



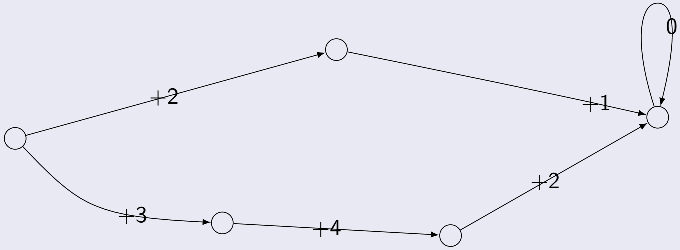
Rollback of Rewards




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

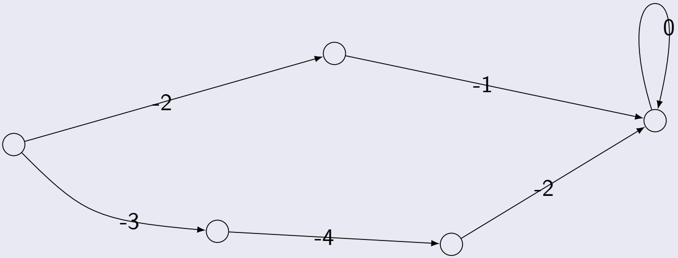
Variations of Reward Tasks I

Maximize reward



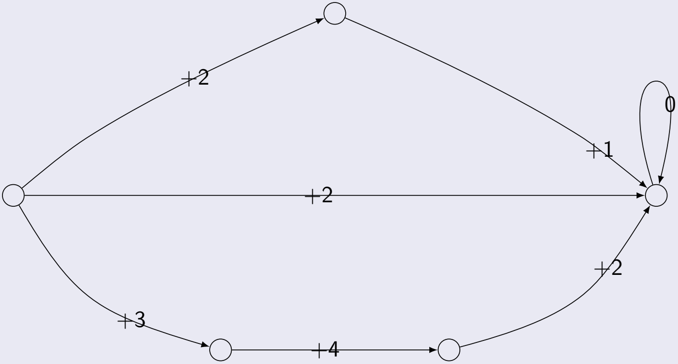
Variations of Reward Tasks II

Minimize penalty

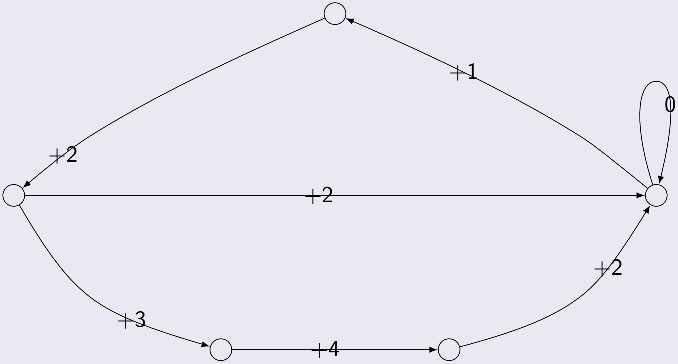


Variations of Reward Tasks III

Path length cedes to reward

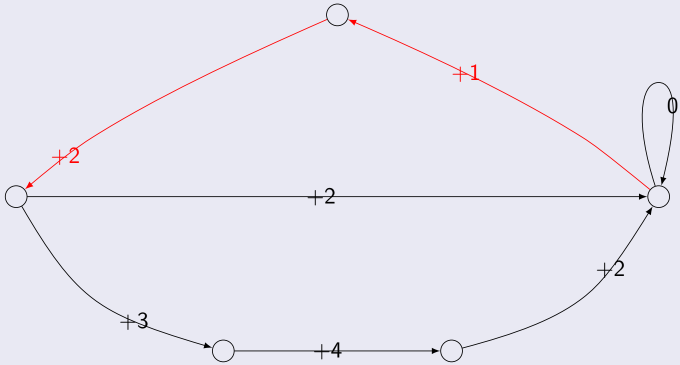


Cycles and positive rewards ϟ



Variations of Reward Tasks V

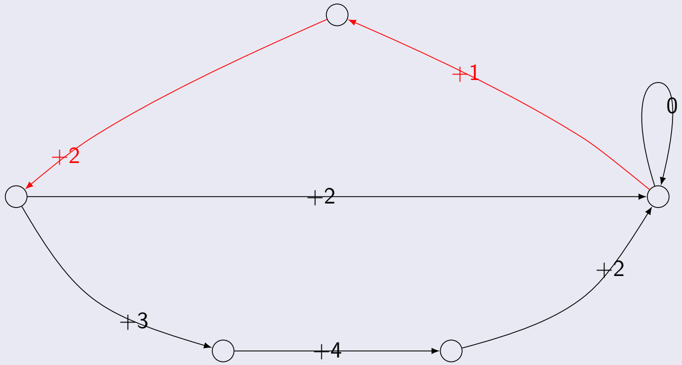
Cycles and positive rewards ↯



Problem: infinite accumulation of reward — **arbitrage!**

Variations of Reward Tasks VI

Cycles and positive rewards ϟ



Problem: infinite accumulation of reward — **arbitrage!**

Solution: discount factor γ

Probabilities and Exploration: Example

Given: n-armed bandit problem. Each trial of some bandit arm, costs $1£$.

Payoff: distributions with differing mean and variance per arm (finite and unknown)

Seeking: strategy with maximum payoff.

Temporal conditions:

- limited
- unlimited
- weighted

time

Exploration/Exploitation Conflict

Problem: finding balance between exploration and exploration leads to conflict

"Greedy" Strategy:
short-term gain — remember "tragedy of the commons"

Note: we dealt with the intricacies of the exploration/exploitation dilemma earlier (UCT); here use only simple strategies

Utility Estimation

Assumption: reward for an action a is random variable with mean $U^*(a)$.

Choose: sequence of actions $a_1, a_2, \ldots, a_t, \ldots$. Then obtain estimate for $U^*(a)$ via

$$U_t(a) = \frac{r_{t_1^a} + \cdots + r_{t_{k_a}^a}}{k_a}$$

where the r at the times $t^a_{i=1,\ldots,k_a}$ which are the times between 1 and t where action a is chosen.

Initialization: $U_0(a)$ is initially set to any value, e.g. 0.

Simple Strategies

Greedy Strategy: choose action

$$a^* = \operatorname*{argmax}_{a \in \mathcal{A}} U_t(a)$$

Alternative: ϵ-Greedy — with probability $1 - \epsilon$, choose greedy, with probability ϵ random action for more exploration

Incremental Estimates

Remark (Incremental Computation of $U_t(a)$): consider t actions having been made and the action at time $t+1$ to be $a_{t+1} = a$. Then

$$U_{t+1}(a) = \frac{1}{k_a+1}\sum_{i=1}^{k_a+1} r_{t_i^a} \qquad \text{(note: } t^a_{k_a+1} = t+1\text{)}$$

$$= \underbrace{U_t(a)}_{\text{old value}} + \underbrace{\frac{1}{k_a+1}}_{t\text{-dependent factor}} \cdot \underbrace{(r_{t+1} - U_t(a))}_{\text{deviation from target value}}$$

Notes:

1. need only to store $U_t(a)$ and k_a
2. the update $w_{t+1} = w_t + \alpha(t)(x_{t+1} - w_t)$ is very common in learning systems
3. $U_{t+1}(a') = U_t(a')$ for $a' \neq a_{t+1}$

Nonstationary Environments: give more importance to recent values, e.g.

$$U_{t+1}(a = a_{t+1}) = U_t(a) + \underbrace{\alpha}_{\text{e.g. constant in } [0,1)} [r_{t+1} - U_t(a)]$$

Reinforcement Learning: Preliminary Definitions

Def. (state): a full description of the current situation, agent and world

Def. (policy): A *policy* π_t at a time t is a conditional probability that an agent in a state s chooses an action a:

$$P(a_t = a \mid s_t = s) = \pi_t(s, a)$$

The Full Reinforcement Learning Problem

Agent: at a time step t it has access to

- current state s_t
- reward just obtained r_t
- current policy π_t

From This: calculate current action choice a_t and following policy π_{t+1}.

Markovian Decision Process

Note:
- full access to current state
- border between agent and environment given by *absolute control*, not by limitation of knowledge

Note: goals of agent specified by rewards r_t

Goal: long-term maximization of cumulated rewards

Example

Examples: reward structure as follows

1. robot is supposed to learn to move: $r_t = 1$ if step ahead
2. maze: 0 per step, 1 for a step outside the maze
3. maze: -1 per step in the maze

Objective

Question: have sequence of rewards $r_{t+1}, r_{t+2}, r_{t+3}, \ldots$. What do we want to maximize?

In General: want to maximize total payoff R_t

2 Cases:

Episodic Tasks: tasks with natural end time T:

$$R_t = r_{t+1} + \cdots + r_T$$

Unlimited Tasks: "$T \to \infty$" require *discounting*

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \\ &= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \end{aligned}$$

with $\gamma \in [0, 1)$

Value Function

Value Function: a measure how good it is for an agent to be in a certain state. This depends on future actions (more precisely, on policy)

$$V^{\pi}(s) = \mathbf{E}_{\pi}(R_t \mid s_t = s) = \mathbf{E}_{\pi}\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right)$$

Q-Function: a measure how good it is for an agent to be in a state and picking a certain action (again dependent on the policy in the following states)

$$Q^{\pi}(s,a) = \mathbf{E}_{\pi}(R_t \mid s_t = s, a_t = a) = \mathbf{E}_{\pi}\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right)$$

Bellmann Equation

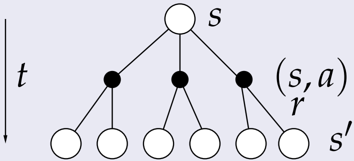
Theorem: with

$$\mathcal{P}^a_{ss'} := P(s_{t+1} = s' \mid s_t = s, a_t = a)$$
$$\mathcal{R}^a_{ss'} := \mathbf{E}(r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s')$$

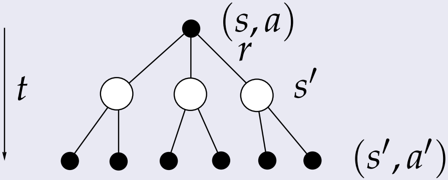
one has

$$V^\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(s,a) \sum_{s' \in \mathrm{Succ}(s)} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right)$$

Backup Diagram for V:



Backup Diagram for Q:



Optimal Policy

Comparison of Policies: we say that $\pi \geq \pi'$ (π is at least as good as π') if for all states s we have $V^{\pi}(s) \geq V^{\pi'}(s)$.

Theorem: there exists always an **optimal** policy π^*, i.e. $\pi^* \geq \pi$ for all policies π.

Note: an optimal policy is not necessarily unique! But V^{π^*} is the same for all optimal policies. Therefore write V^* for optimal value function.

Analogously: define $Q^*(s,a)$

Remark: one has

$$Q^*(s,a) = \mathbf{E}(r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a)$$

Bellmanns Optimality Criterion

Bellmann's Optimality Equation: one has

$$V^*(s) = \max_a \mathbf{E}(r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a)$$

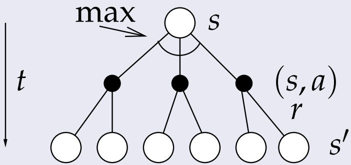
Analogously:

$$Q^*(s,a) = \mathbf{E}\left(r_{t+1} + \gamma \max_{a' \in \mathcal{A}(s_{t+1})} Q^*(s_{t+1}, a') \mid s_t = s, a_t = a\right)$$

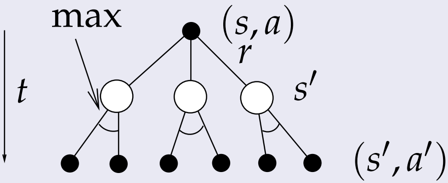
Backup Diagrams

(Sutton and Barto 1998)

For V^*:



For Q^*:



Methods:

- dynamic programming
- value iteration
- Q-learning

Q-Learning

Q-Learning: update rule given by

$$Q(s_t, a_t) \longleftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_{a \in \mathcal{A}(s_{t+1})} Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

Theorem (Watkins): if all (s, a) are being update often enough, Q converges towards Q^*, independently of policy π.

Q-Learning: Properties

Advantages:

- off-policy
- does not require explicit averaging — done implicitly, as you go
- no model required

Reinforcement Learning: uses Q-Learning as central model — many variants and improvements exist.

General Remarks

Reinforcement Learning: learning from **delayed** rewards

In Particular: Q-Learning requires

- no model of dynamics
- only immediate backup
- but: state must have

  Markov Property: the result of an action must only depend on a current state variable which must be known to the agent. In particular

  1. it must not depend on some "memory" effects unseen by the agent
  2. it must not depend on the history of the agent or the world

Theory and Practice of Artificial Intelligence

Probabilities

Daniel Polani

School of Computer Science
University of Hertfordshire

March 9, 2017

Motivation (Fermat, Pascal)

Consider: pot of money, two-player bet (coin toss)

Problem: game is interrupted — what is fair split of pot *before* coin toss?

Idea:
- assume none of the 2 outcomes of coin toss preferred
- associate a *probability* of $1/2$ with each of the outcomes
- this is the weight by which the payoff of the pot is multiplied for each of the potential outcomes: each of the players gets $1/2$ of the pot

Discussion

Note:

- this is a special case
- in general, the probabilities are not identical for the outcomes
- also, more than 2 outcomes possible

Def.: **random variables** and **probabilities**

- a **random variable** X is an object with potential outcomes $x_1, x_2, \ldots$ from a set $\mathcal{X}$;
- each of these outcomes $x_1, x_2, \ldots$ is associated with a real value, its **probability** $P(X = x_1) \equiv p(x_1), P(X = x_2) \equiv p(x_2), \dots \in \mathbb{R}$ s.t.
  1. $p(x) \geq 0$ for all $x \in \mathcal{X}$;
  2. $\sum_{x \in \mathcal{X}} p(x) = 1$

Example: Die

Consider:

1. die with 6 sides;
2. assume no reason to assume asymmetry, i.e. no side is preferred
   (**Laplace's principle of insufficient reason**)
3. consider outcomes D, probability of each outcome 1,2,...,6 is $P(D = 1) = P(D = 2) = \cdots = P(D = 6) = 1/6$.

Joint Variables

Example: two dice, described by random variables
Joint Variables: (D_1, D_2) with probabilities $p(d_1, d_2) = ?$

Joint Variables/Probabilities

Example: two dice, described by random variables

Joint Variables: (D_1, D_2) with probabilities
$p(d_1, d_2) = 1/36 = 1/6 \cdot 1/6$

Outcomes: all combinations of $\mathcal{D} \times \mathcal{D}$, with $\mathcal{D} = \{1, \ldots, 6\}$ with equal probability

Note: we will see, this is a special case!

Example: Sister/Brother

Consider: random variables $C_1, C_2 \in \{\texttt{boy}, \texttt{girl}\}$, the first and the second child of a family.

Question: assuming that the first child is a girl ($c_1 = \texttt{girl}$), what is the probability that the second child is a boy?

Example: Sister/Brother

Consider: random variables $C_1, C_2 \in \{\texttt{boy}, \texttt{girl}\}$, the first and the second child of a family.

Question: assuming that the first child is a girl ($c_1 = \texttt{girl}$), what is the probability that the second child is a boy?

Answer: consider outcomes:

c_1	c_2	$p(c_1, c_2)$
boy	boy	1/4
boy	girl	1/4
girl	boy	1/4
girl	girl	1/4

Answer: Sister/Brother

Remember Question: **assuming first child is a girl**

Answer: consider outcomes:

c_1	c_2	$p(c_1, c_2)$
boy	boy	1/4
boy	girl	1/4
girl	boy	1/4
girl	girl	1/4

Note: only cases considered with $c_1 = \texttt{girl}$

Here: total weight (probability) of cases with $C_1 = \texttt{girl}$ is 1/2, and 1/4 of which cover the cases where $C_2 = \texttt{boy}$.

Hence: probability that second child is boy if first child is girl given by

$$\frac{1/4}{1/2} = 1/2$$

Example: Sister/Brother II

Question II: assume **one of the children** is a girl, what is the probability that the other child is a boy?

Example: Sister/Brother II

Question II: assume **one of the children** is a girl, what is the probability that the other child is a boy?

Answer: consider outcomes:

c_1	c_2	$p(c_1, c_2)$
boy	boy	$1/4$
boy	girl	$1/4$
girl	boy	$1/4$
girl	girl	$1/4$

Note:
- now total probability of the cases to consider ($C_1 = \texttt{girl}$ or $C_2 = \texttt{girl}$) is 3/4!
- in 2 of these cases, the other child is a boy, original probability is now $1/4+1/4=1/2$

Hence: probability that other child is boy if one child is girl is:

$$\frac{1/2}{3/4} = 2/3$$

Conditional Probabilities

Def.: probability of a random variable Y if another random variable X is given is called **conditional probability**, and written $P(Y = y|X = x) \equiv p(y|x)$.

Example: in boy/girl example, we calculated in

1. Question I: $P(C_2 = \texttt{boy}|C_1 = \texttt{girl})$
2. Question II: $P(C_1 = \texttt{boy}\, or\, C_2 = \texttt{boy}|C_1 = \texttt{girl}\, or\, C_2 = \texttt{girl})$

Joint Probabilities and Marginalization

Summary:

- joint distribution of two variables C_1, C_2: $p(c_1, c_2)$
- probability of only one of the variables obtained by **marginalization** — sum over the other:

$$p(c_1) = \sum_{c_2 \in \mathcal{C}_2} p(c_1, c_2)$$

- example for marginalization (over the second variable): probability of **first child being girl** (was: $1/2$)
- **not marginalization**: probability of **one child being girl** — for this, we had to consider both first and second child!

Conditional Probabilities

Summary:

- conditional distribution of a variable C_2 given another variable C_1: $p(c_2|c_1)$
- example: probability of second child being boy if first child is girl, expressed from the joint distribution:

$$P(C_2 = \texttt{boy}|C_1 = \texttt{girl}) = \frac{P(C_1 = \texttt{girl}, C_2 = \texttt{boy})}{P(C_1 = \texttt{girl})}$$

- note: select all (and only) cases where condition $C_1 = \texttt{girl}$ holds. Take probability of desired case $C_2 = \texttt{boy}$ and normalize by the total probability of the condition $C_1 = \texttt{girl}$.

Conditional Probabilities II

Note: conditional can be computed also in more general cases, namely when outcomes are combined (Question II)
Divide probability of desired outcome through total probability of conditional, in general:

$$p(y|x) = \frac{p(x,y)}{p(x)}$$

Note: in Question II, the notation for the combined conditions (one of the children is a girl/boy) would have to be suitably denoted, but simplified things by summing up the probabilities of all relevant cases.

Bayes' Theorem

Remark: Note that

$$p(d_2|d_1)p(d_1) = p(d_1, d_2) = p(d_1|d_2)p(d_2)\,.$$

From this follows

Bayes' Theorem: One has:

$$p(d_2|d_1) = \frac{p(d_1|d_2)p(d_2)}{p(d_1)}\,.$$

Note:

- Bayes' Theorem highly important: allows one to turn around the direction of a conditional. If conditional in one direction is known, the other can be inferred.
- Sufficient to compute $p(d_1|d_2)p(d_2)$; denominator obtained by normalization.

Theory and Practice of Artificial Intelligence

Probabilistic Inference

Daniel Polani

School of Computer Science
University of Hertfordshire

March 9, 2017

Simplified Minesweeper

Example

1. Assume a mine is located at one of positions $1 \ldots 5$.
2. Its presence at position m is detectable at positions $m + 1$ and $m - 1$ (if in the field).
3. **Task:** Solve the minesweeper problem — find the mine.

Simplified Minesweeper: Bayes Model

Approach

1. $p(m)$ is the probability that the mine is at location m.
2. If we do not know anything, by symmetry: $p(m) = 1/5$.
3. A detection d can assume the values $\{\texttt{boom!}, \texttt{detect}, \texttt{nothing}\}$.
4. At position i, assuming the mine is at m, we have detection probability $p(d|m,i)$ with:

$$p(d|m,i) := \begin{cases} 1 & \text{if } d = f(m,i) \\ 0 & else \end{cases}$$

where

$$f(m,i) = \begin{cases} \texttt{boom!} & \text{if } m = i \\ \texttt{detect} & \text{if } |i-m| = 1 \\ \texttt{nothing} & \text{else}. \end{cases}$$

Simplified Minesweeper: $p(d|m,i)$

`nothing`

m \ i	1	2	3	4	5
1	0	0	1	1	1
2	0	0	0	1	1
3	1	0	0	0	1
4	1	1	0	0	0
5	1	1	1	0	0

`detect`

m \ i	1	2	3	4	5
1	0	1	0	0	0
2	1	0	1	0	0
3	0	1	0	1	0
4	0	0	1	0	1
5	0	0	0	1	0

`boom!`

m \ i	1	2	3	4	5
1	1	0	0	0	0
2	0	1	0	0	0
3	0	0	1	0	0
4	0	0	0	1	0
5	0	0	0	0	1

Simplified Minesweeper: Bayes' Theorem

1. Assume: uncovering position $i = 2$, we observe $d = \texttt{nothing}$.

2. Note that for each m, because of Bayes:

$$p(m|d, i=2) = \frac{1}{Z} p(d|m, i=2) p(m)$$

$$= Z^{-1} \left[p(d|\underset{\substack{\parallel \\ 1}}{m}, i) \cdot \frac{1}{5}, p(d|\underset{\substack{\parallel \\ 2}}{m}, i) \cdot \frac{1}{5}, p(d|\underset{\substack{\parallel \\ 3}}{m}, i) \cdot \frac{1}{5}, p(d|\underset{\substack{\parallel \\ 4}}{m}, i) \cdot \frac{1}{5}, p(d|\underset{\substack{\parallel \\ 5}}{m}, i) \cdot \frac{1}{5} \right]_{i=2}$$

3. Reminder

$$p(d|m,i) := \begin{cases} 1 & \text{if } d = f(m,i) \\ 0 & \text{else,} \end{cases} \qquad f(m,i) = \begin{cases} \texttt{boom!} & \text{if } m = i \\ \texttt{detect} & \text{if } |i-m| = 1 \\ \texttt{nothing} & \text{else.} \end{cases}$$

4.

$$\begin{aligned} p(m|d, i=2) &= Z^{-1} \cdot [0, 0, 0, 0.2, 0.2] \\ &= [0, 0, 0, 0.5, 0.5] \end{aligned} \qquad (1)$$

Simplified Minesweeper: Bayes' Theorem

1. Assume: uncovering position $i = 2$, we observe $d = \texttt{detect}$.
2. For each m (Bayes):

$$p(m|d, i = 2) = \frac{1}{Z} p(d|m, i = 2)p(m)$$

$$= Z^{-1} \left[p(d|m, i = 2)\Big|_{m=1} \cdot \frac{1}{5}, p(d|m, i = 2)\Big|_{m=2} \cdot \frac{1}{5}, p(d|m, i = 2)\Big|_{m=3} \cdot \frac{1}{5}, p(d|m, i = 2)\Big|_{m=4} \cdot \frac{1}{5}, p(d|m, i = 2)\Big|_{m=5} \cdot \frac{1}{5} \right]$$

3. Reminder

$$p(d|m, i) := \begin{cases} 1 & \text{if } d = f(m, i) \\ 0 & \text{else} \end{cases} \quad f(m, i) = \begin{cases} \texttt{boom!} & \text{if } m = i \\ \texttt{detect} & \text{if } |i - m| = 1 \\ \texttt{nothing} & \text{else.} \end{cases}$$

4.

$$\begin{aligned} p(m|d, i = 2) &= Z^{-1} \cdot [0.2, 0, 0.2, 0, 0] \\ &= [0.5, 0, 0.5, 0, 0] \end{aligned} \tag{2}$$

Simplified Minesweeper: Bayes' Theorem

1. Assume : uncovering position $i = 1$, we observe $d =$ `detect`.
2. For each m (Bayes):

$$p(m|d, i=1) = \frac{1}{Z} p(d|m, i=1) p(m)$$
$$= Z^{-1} \left[p(d|m, i=1)\big|_{m=1} \cdot \frac{1}{5}, p(d|m, i=1)\big|_{m=2} \cdot \frac{1}{5}, p(d|m, i=1)\big|_{m=3} \cdot \frac{1}{5}, \right.$$
$$\left. p(d|m, i=1)\big|_{m=4} \cdot \frac{1}{5}, p(d|m, i=1)\big|_{m=5} \cdot \frac{1}{5} \right]$$

3. Reminder

$$p(d|m,i) := \begin{cases} 1 & \text{if } d = f(m,i) \\ 0 & \text{else} \end{cases} \quad f(m,i) = \begin{cases} \texttt{boom!} & \text{if } m = i \\ \texttt{detect} & \text{if } |i-m| = 1 \\ \texttt{nothing} & \text{else.} \end{cases}$$

4.

$$\begin{aligned} p(m|d, i=1) &= Z^{-1} \cdot [0, 0.2, 0, 0, 0] \\ &= [0, 1, 0, 0, 0] \end{aligned} \tag{3}$$

Homework

Wumpus World

Check (Russell and Norvig 2002) **Wumpus World Revisited**, probabilistic treatment of the wumpus world.

Dude, Where Is My Car? (Monty Hall Problem)

Blackboard/Practical

Monty Hall (Python) I

```
from random import *
from rational import Rational

# in-line operation - rare use of side effect

def normalize(prob):
    Z = float(sum(prob[outcome]
                  for outcome in prob))
    for outcome in prob:
        prob[outcome] /= Z

# main

p = {}

p["cso"] = {}          # a joint probability for c,s,o, in
    this order
```

Monty Hall (Python) II

```
for run in xrange(1000000):
    doors = set(range(1,4))
    car = sample(doors,1)[0]          # sample and pick
    select = sample(doors,1)[0]

    open_door = sample(doors - set([car, select]),
        1).pop()

    outcome = (car, select, open_door)
    if not outcome in p["cso"]:
        p["cso"][outcome] = 0
    p["cso"][outcome] += 1

# normalize

normalize(p["cso"])
```

Monty Hall (Python) III

```
# seek: probability of where car is, conditioned on
    observed moves

# for this, we need p(s,o)

p["so"] = {}

for car, select, open_door in p["cso"]:
    observation = select, open_door
    if not observation in p["so"]:
        p["so"][observation] = 0
    p["so"][observation] += p["cso"][car, select,
        open_door]

normalize(p["so"])

# conditional
```

Monty Hall (Python) IV

```
p["c|so"] = {}

for car, select, open_door in p["cso"]:
    outcome = car, select, open_door
    p["c|so"][outcome] = p["cso"][outcome] /
        p["so"][select, open_door]

for car, select, open_door in p["c|so"]:
    if (select, open_door) == (1, 2):
        print car, p["c|so"][car, select, open_door]
```

Alternative Probability Classes (sampler.py)

```python
class Sampler:
    def __init__(self):
        self.n = {}

    def __getitem__(self, x):
        return self.n[x]

    def observe(self, x):
        n = self.n
        if not x in n: n[x] = 0
        n[x] += 1
```

Alternative Probability Classes (prob2.py)

```
from sampler import *

class Prob:
    def __init__(self, sampler = None):
        if not sampler:
            self.p = {}
        else:
            self.p = dict((x, sampler[x]) for x in
                sampler.n)
        self.normalized = False

    def __getitem__(self, x):
        self.normalize()
        return self.p[x]

    def __setitem__(self, x, p):
        self.p[x] = p
        self.normalized = False
```

Usage of Alternative Probability Classes

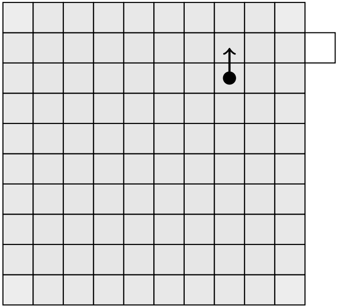
```
from prob2 import *
import random

sampler = Sampler()

for i in xrange(100000):
    die = random.randint(1,6)
    sampler.observe(die)

p = Prob(sampler)


print p
```

Robot

- can move
- can turn
- can only detect `empty`, `wall`, `door` in front

Localization

- has knowledge about the map, but
- unknown position
- unknown orientation



Example: Markovian Self-Localization

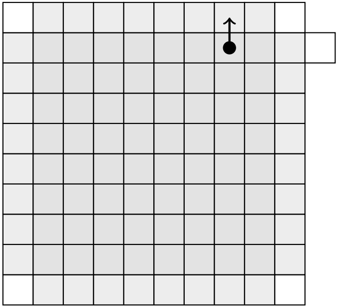
(Thrun et al. 2005)

Robot

- can move
- can turn
- can only detect `empty`, `wall`, `door` in front

Localization

- has knowledge about the map, but
- unknown position
- unknown orientation



Example: Markovian Self-Localization

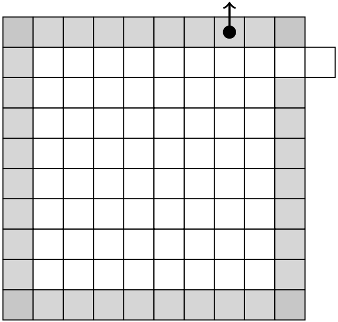
(Thrun et al. 2005)

Robot

- can move
- can turn
- can only detect `empty`, `wall`, `door` in front

Localization

- has knowledge about the map, but
- unknown position
- unknown orientation



Example: Markovian Self-Localization

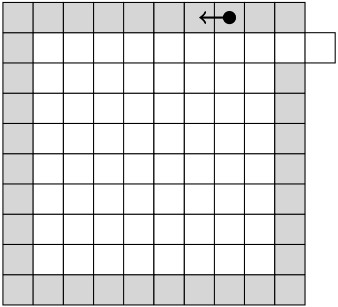
(Thrun et al. 2005)

Robot

- can move
- can turn
- can only detect `empty`, `wall`, `door` in front

Localization

- has knowledge about the map, but
- unknown position
- unknown orientation



Example: Markovian Self-Localization

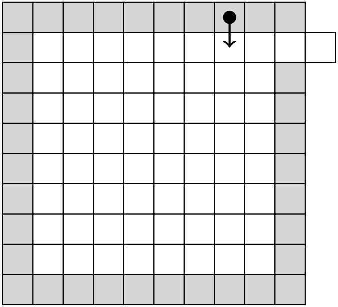
(Thrun et al. 2005)

Robot

- can move
- can turn
- can only detect `empty`, `wall`, `door` in front

Localization

- has knowledge about the map, but
- unknown position
- unknown orientation



Example: Markovian Self-Localization

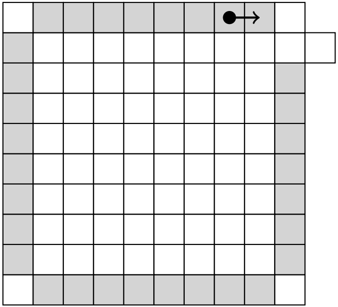
(Thrun et al. 2005)

Robot

- can move
- can turn
- can only detect `empty`, `wall`, `door` in front

Localization

- has knowledge about the map, but
- unknown position
- unknown orientation



Example: Markovian Self-Localization

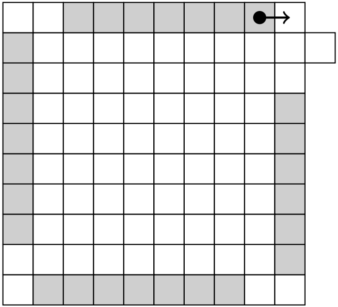
(Thrun et al. 2005)

Robot

- can move
- can turn
- can only detect `empty`, `wall`, `door` in front

Localization

- has knowledge about the map, but
- unknown position
- unknown orientation



Example: Markovian Self-Localization

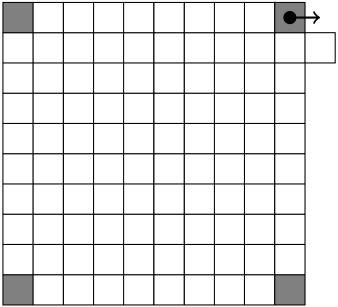
(Thrun et al. 2005)

Robot

- can move
- can turn
- can only detect `empty`, `wall`, `door` in front

Localization

- has knowledge about the map, but
- unknown position
- unknown orientation



Example: Markovian Self-Localization

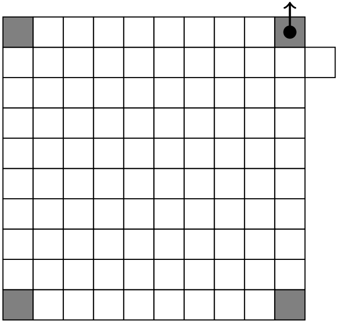
(Thrun et al. 2005)

Robot

- can move
- can turn
- can only detect `empty`, `wall`, `door` in front

Localization

- has knowledge about the map, but
- unknown position
- unknown orientation



Example: Markovian Self-Localization

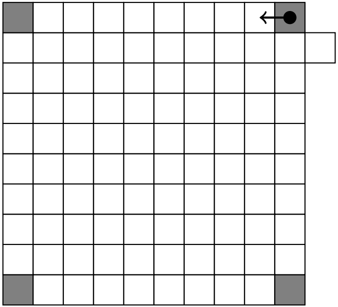
(Thrun et al. 2005)

Robot

- can move
- can turn
- can only detect `empty`, `wall`, `door` in front

Localization

- has knowledge about the map, but
- unknown position
- unknown orientation



Example: Markovian Self-Localization

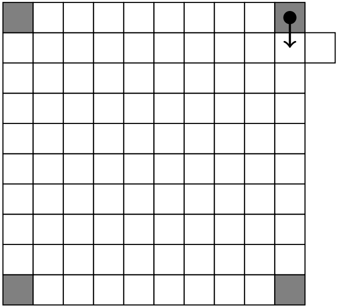
(Thrun et al. 2005)

Robot

- can move
- can turn
- can only detect `empty`, `wall`, `door` in front

Localization

- has knowledge about the map, but
- unknown position
- unknown orientation



Example: Markovian Self-Localization

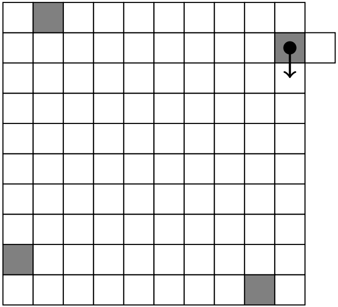
(Thrun et al. 2005)

Robot

- can move
- can turn
- can only detect `empty`, `wall`, `door` in front

Localization

- has knowledge about the map, but
- unknown position
- unknown orientation



Example: Markovian Self-Localization

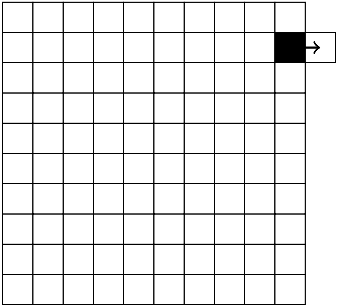
(Thrun et al. 2005)

Robot

- can move
- can turn
- can only detect `empty`, `wall`, `door` in front

Localization

- has knowledge about the map, but
- unknown position
- unknown orientation



Procedure

Preparation

1. consider set of states: $\mathcal{S} = \mathcal{R} \times \mathcal{D}$ with
   - location r
   - direction (orientation) d
   - together: state $s = (r, d)$
2. consider possible observations $o \in \mathcal{O} = \{\texttt{empty}, \texttt{wall}, \texttt{door}\}$
3. consider possible actions $a \in \mathcal{A} = \{\texttt{move}, \texttt{turn}\}$
4. define probability $p(o|s)$ that an observation o will be made if the state is s
5. define probability $p(s'|s, a)$ that the new state (location, orientation) of the robot will be s' if old state was s and action a is taken.
   more specifically: if action a is deterministic with $a(s) = s'$, then set
   $$p(s'|s, a) := \begin{cases} 1 & \text{if } a(s) = s' \\ 0 & \text{else .} \end{cases}$$

Procedure II

Algorithm

1. start with probability prior $p(s)$ as the probability that the robot is at state (location, orientation) s
2. with an observation o, update the robot location probability with Bayes:

$$p(s|o) := \frac{1}{Z} p(o|s) p(s)$$

   where Z guarantees normalization, i.e. that $\sum_s p(s|o) = 1$. The posterior $p(s|o)$ becomes the new state probability of the robot after observation; call this $p(\tilde{s})$
3. update for the move (action) of the robot. The probability for the new state of the robot after performing action a is

$$p(s'|a) := \sum_{\tilde{s}} p(s'|\tilde{s}, a) p(\tilde{s})$$

   After choice of a, this distribution of states is our new model of what state the robot is in.
4. Use this as the new prior $p(s)$ and repeat loop from step 1

Notes

- steps 1 and 2 are the usual Bayesian update
- step 3 is additionally required to deal with movement of the robot

Part II

References

References I

Auer, P., (2003). Using Confidence Bounds for Exploitation-exploration Trade-offs. *J. Mach. Learn. Res.*, 3:397–422.

`http://dl.acm.org/citation.cfm?id=944919.944941`, Feb 2017

Bradberry, J., (2015). Introduction to Monte Carlo Tree Search.

`Introduction to Monte Carlo Tree Search`, 8. Feb. 2017

Browne, C., (2012). Monte Carlo Tree Search.

`https://pdfs.semanticscholar.org/0c6b/1e96cb05a266b410d25547dbf6bbc26ff0eb.pdf`, 8. Feb. 2017

Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S., (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1).

References II

Kocsis, L., and Szepesvári, C., (2006). Bandit Based Monte-carlo Planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, 282–293. Berlin, Heidelberg: Springer-Verlag.
`http://dx.doi.org/10.1007/11871842_29`, Feb 2017

Pearl, J., (1984). *Heuristics: Intelligent Search Strategies for Computer Problem-Solving*. Addison Wesley.

Russell, S., and Norvig, P., (2002). *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Prentice Hall. Second edition.

Saygin, A., Cicekli, I., and Akman, V., (2000). Turing Test: 50 Years Later. *Minds and Machines*, 10(4):463–518.

Sedgewick, R., and Wayne, K., (2011). *Algorithms, 4th Edition*. Addison-Wesley.

References III

Sutton, R. S., and Barto, A. G., (1998). *Reinforcement Learning*. Cambridge, Mass.: MIT Press.

Thrun, S., Burgard, W., and Fox, D., (2005). *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents series)*. Intelligent robotics and autonomous agents. The MIT Press.