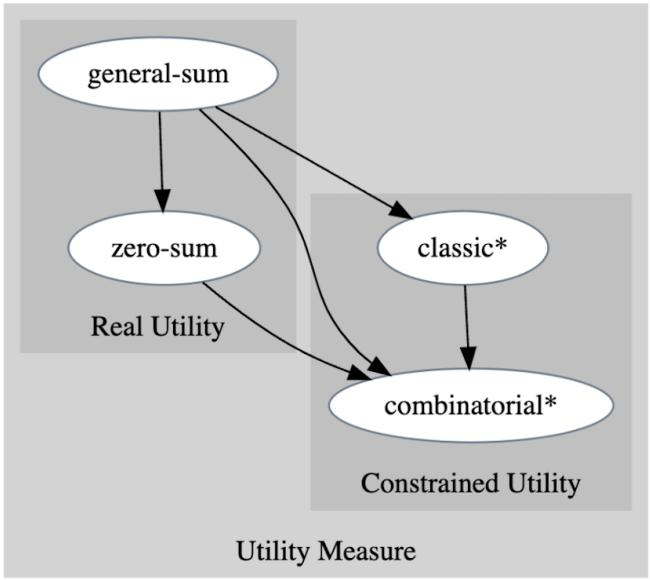
Computational Techniques in Game-Theoretic Systems Pushing the Solving Horizon

Max Fierro

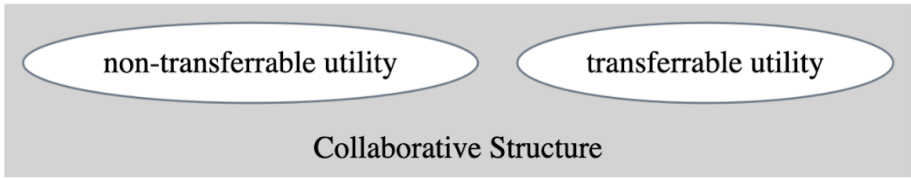
Department of EECS
University of California, Berkeley
Berkeley, CA
maxfierro@berkeley.edu

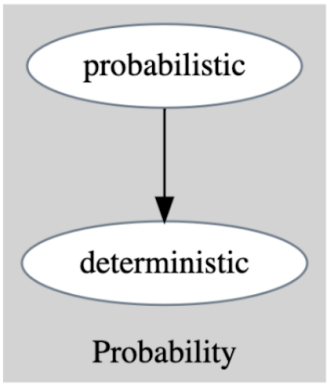
April 20, 2024

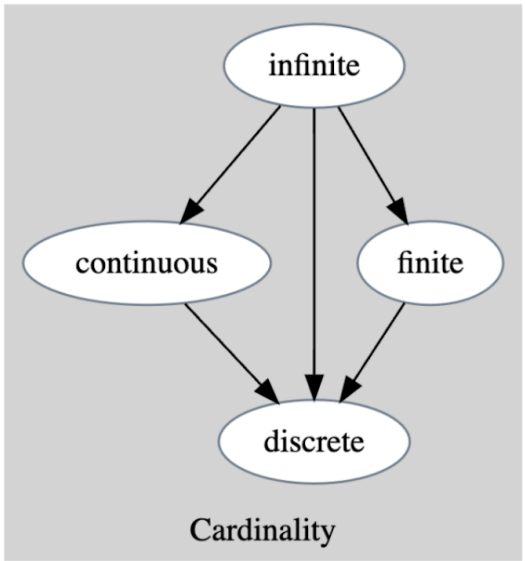
Objectives

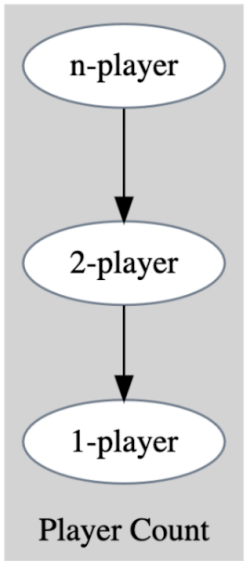


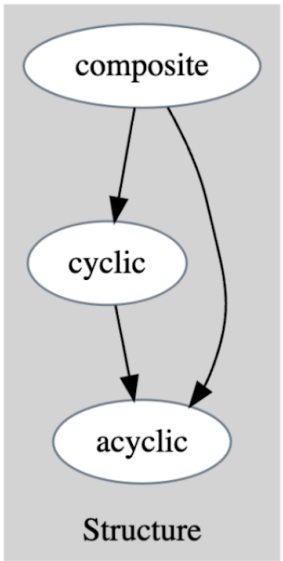
A “game” or “ruleset” can be...

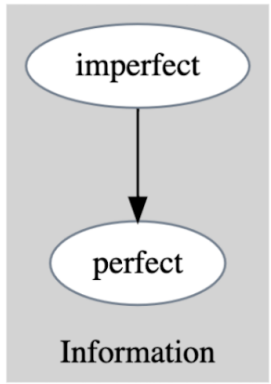












Objectives

- Today we will mainly talk about **search**, which encompasses all forms of automatic traversal of games as defined by their rules.
- We will not refer to any single type of game; we will speak about **computational search techniques** as they broadly apply to games with discrete representations (!) which allow us to perform search on a greater number of "positions."
- Keeping away from deep systems concepts, we will motivate:
  - How strong and weak solutions to "big" games are computed and stored in realistic space and time.
  - How to compute a "solution set" that allows us to ask questions about meta-strategies and game structure.
  - How you can grab a computer and strongly solve your favorite game (hopefully not taking the fun out of it in the process).

Objectives

- Today we will mainly talk about **search**, which encompasses all forms of automatic traversal of games as defined by their rules.
- We will not refer to any single type of game; we will speak about **computational search techniques** as they broadly apply to games with discrete representations (!) which allow us to perform search on a greater number of "positions."
- Keeping away from deep systems concepts, we will motivate:
  - How strong and weak solutions to "big" games are computed and stored in realistic space and time.
  - How to compute a "solution set" that allows us to ask questions about meta-strategies and game structure.
  - How you can grab a computer and strongly solve your favorite game (hopefully not taking the fun out of it in the process).

Objectives

- Today we will mainly talk about **search**, which encompasses all forms of automatic traversal of games as defined by their rules.
- We will not refer to any single type of game; we will speak about **computational search techniques** as they broadly apply to games with discrete representations (!) which allow us to perform search on a greater number of "positions."
- Keeping away from deep systems concepts, we will motivate:
  - How strong and weak solutions to "big" games are computed and stored in realistic space and time.
  - How to compute a "solution set" that allows us to ask questions about meta-strategies and game structure.
  - How you can grab a computer and strongly solve your favorite game (hopefully not taking the fun out of it in the process).

Objectives

- Today we will mainly talk about **search**, which encompasses all forms of automatic traversal of games as defined by their rules.
- We will not refer to any single type of game; we will speak about **computational search techniques** as they broadly apply to games with discrete representations (!) which allow us to perform search on a greater number of "positions."
- Keeping away from deep systems concepts, we will motivate:
  - How strong and weak solutions to "big" games are computed and stored in realistic space and time.
  - How to compute a "solution set" that allows us to ask questions about meta-strategies and game structure.
  - How you can grab a computer and strongly solve your favorite game (hopefully not taking the fun out of it in the process).

Objectives

- Today we will mainly talk about **search**, which encompasses all forms of automatic traversal of games as defined by their rules.
- We will not refer to any single type of game; we will speak about **computational search techniques** as they broadly apply to games with discrete representations (!) which allow us to perform search on a greater number of "positions."
- Keeping away from deep systems concepts, we will motivate:
  - How strong and weak solutions to "big" games are computed and stored in realistic space and time.
  - How to compute a "solution set" that allows us to ask questions about meta-strategies and game structure.
  - How you can grab a computer and strongly solve your favorite game (hopefully not taking the fun out of it in the process).

Objectives

- Today we will mainly talk about **search**, which encompasses all forms of automatic traversal of games as defined by their rules.
- We will not refer to any single type of game; we will speak about **computational search techniques** as they broadly apply to games with discrete representations (!) which allow us to perform search on a greater number of "positions."
- Keeping away from deep systems concepts, we will motivate:
  - How strong and weak solutions to "big" games are computed and stored in realistic space and time.
  - How to compute a "solution set" that allows us to ask questions about meta-strategies and game structure.
  - How you can grab a computer and strongly solve your favorite game (hopefully not taking the fun out of it in the process).

Overview

We will cover three high-level aspects high-performance search:

- **Representation:** How do we differentiate games we care about in a convenient yet abstract way?
- **Computation:** How do we generate the information we want to know from representations?
- **Storage:** How do we store and retrieve the information we generate during computation?

Representation

- For simplicity, we will consider the complete-information (!) and deterministic (!) case of discrete games.
- An **extensive-form game** is 4-tuple $G = \langle N, H, p, (\succsim_i) \rangle$ where [1],
  - N is the set of players, usually $\{1, \ldots, n\}$ in an n-player game,
  - H is a set of sequences (or "histories") whose prefixes are all in H,
  - $p : H \to N$ assigns a player to each non-terminal history,
  - player $i \in N$ has a preference relation $\succsim_i$ on the set $Z \subseteq H$ of terminal histories (which is reflexive and transitive).

Representation

- For simplicity, we will consider the complete-information (!) and deterministic (!) case of discrete games.
- An **extensive-form game** is 4-tuple $G = \langle N, H, p, (\succsim_i) \rangle$ where [1],
  - N is the set of players, usually $\{1, \ldots, n\}$ in an n-player game,
  - H is a set of sequences (or "histories") whose prefixes are all in H,
  - $p : H \to N$ assigns a player to each non-terminal history,
  - player $i \in N$ has a preference relation $\succsim_i$ on the set $Z \subseteq H$ of terminal histories (which is reflexive and transitive).

Representation

- For simplicity, we will consider the complete-information (!) and deterministic (!) case of discrete games.
- An **extensive-form game** is 4-tuple $G = \langle N, H, p, (\succsim_i) \rangle$ where [1],
  - N is the set of players, usually $\{1, \ldots, n\}$ in an n-player game,
  - H is a set of sequences (or "histories") whose prefixes are all in H,
  - $p : H \to N$ assigns a player to each non-terminal history,
  - player $i \in N$ has a preference relation $\succsim_i$ on the set $Z \subseteq H$ of terminal histories (which is reflexive and transitive).

Representation

- For simplicity, we will consider the complete-information (!) and deterministic (!) case of discrete games.
- An **extensive-form game** is 4-tuple $G = \langle N, H, p, (\succsim_i) \rangle$ where [1],
  - N is the set of players, usually $\{1, \ldots, n\}$ in an n-player game,
  - H is a set of sequences (or "histories") whose prefixes are all in H,
  - $p : H \to N$ assigns a player to each non-terminal history,
  - player $i \in N$ has a preference relation $\succsim_i$ on the set $Z \subseteq H$ of terminal histories (which is reflexive and transitive).

Representation

- For simplicity, we will consider the complete-information (!) and deterministic (!) case of discrete games.
- An **extensive-form game** is 4-tuple $G = \langle N, H, p, (\succsim_i) \rangle$ where [1],
  - N is the set of players, usually $\{1, \ldots, n\}$ in an n-player game,
  - H is a set of sequences (or "histories") whose prefixes are all in H,
  - $p : H \to N$ assigns a player to each non-terminal history,
  - player $i \in N$ has a preference relation $\succsim_i$ on the set $Z \subseteq H$ of terminal histories (which is reflexive and transitive).

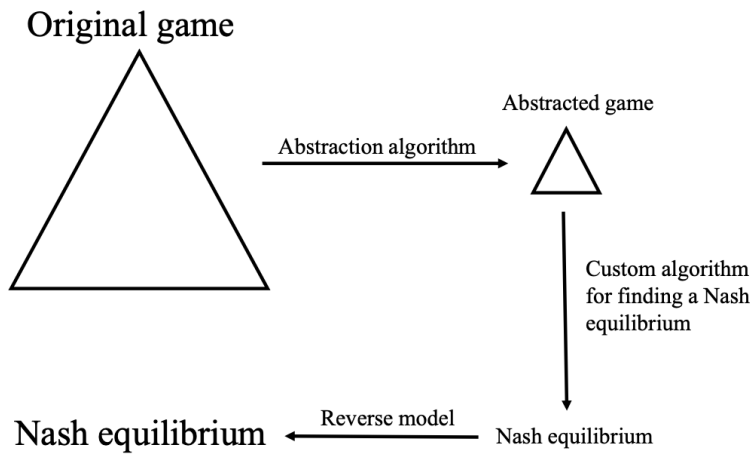
Representation

- For simplicity, we will consider the complete-information (!) and deterministic (!) case of discrete games.
- An **extensive-form game** is 4-tuple $G = \langle N, H, p, (\succsim_i) \rangle$ where [1],
  - N is the set of players, usually $\{1, \ldots, n\}$ in an n-player game,
  - H is a set of sequences (or "histories") whose prefixes are all in H,
  - $p : H \to N$ assigns a player to each non-terminal history,
  - player $i \in N$ has a preference relation $\succsim_i$ on the set $Z \subseteq H$ of terminal histories (which is reflexive and transitive).

Representation

Many histories are often *structurally* similar or even equivalent:

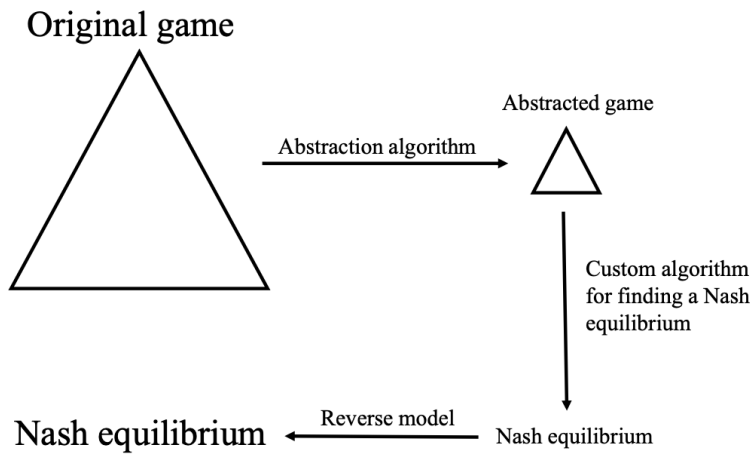
- We establish equivalence classes among histories, creating the intuitive notion of a set S of game "states."
- Histories can be equivalent for many reasons (e.g., game board symmetry or cyclic gameplay).
- In the context of methods for imperfect information analysis, T. Sandholm [2] refers to this process as game abstraction (see illustration).



Representation

Many histories are often *structurally* similar or even equivalent:

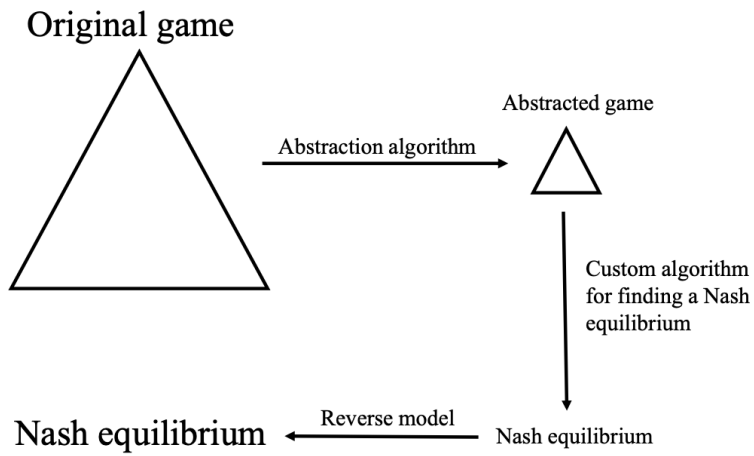
- We establish equivalence classes among histories, creating the intuitive notion of a set S of game "states."
- Histories can be equivalent for many reasons (e.g., game board symmetry or cyclic gameplay).
- In the context of methods for imperfect information analysis, T. Sandholm [2] refers to this process as game abstraction (see illustration).



Representation

Many histories are often *structurally* similar or even equivalent:

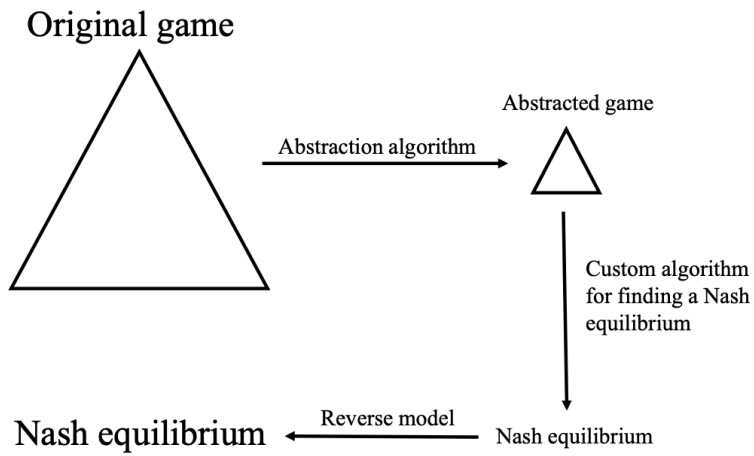
- We establish equivalence classes among histories, creating the intuitive notion of a set S of game "states."
- Histories can be equivalent for many reasons (e.g., game board symmetry or cyclic gameplay).
- In the context of methods for imperfect information analysis, T. Sandholm [2] refers to this process as game abstraction (see illustration).



Representation

Many histories are often *structurally* similar or even equivalent:

- We establish equivalence classes among histories, creating the intuitive notion of a set S of game "states."
- Histories can be equivalent for many reasons (e.g., game board symmetry or cyclic gameplay).
- In the context of methods for imperfect information analysis, T. Sandholm [2] refers to this process as game abstraction (see illustration).



Representation

Revisiting the definition $G = \langle N, H, p, (\succsim_i) \rangle$, we will

- replace H with a set of states S (through some implicit $s : H \to S$),
- and define a "utility" function $u : S \to \mathbb{R}^{|N|}$ to replace $(\succsim_i)$, which satisfies $a \succsim_i b \iff u(s(a))_i \geq u(s(b))_i$ for all histories $a, b \in H$ and players $i \in N$.

Then, we can use the game rules to express $\langle N, S, p, u \rangle$ as an implicit **game graph** in a generic computer program:

```
type State = ? // What is the common meaning of State?

interface Game {
    function utility(State) -> Vector<Utility>
    function transition(State) -> Set<State>
    function turn(State) -> Integer
    function start() -> State
}
```

Computation

Hashing functions allow implementations to transact in "States":

- These functions encode in-memory representations that are nice to work with into dense sequences of bits.
- Their output should be a common bitwidth (generally 64 bits), which is how we can represent them using a single sized type.
- Must be invertible (or "perfect hash functions"), so they are usually custom-made for each game.
- High-performance implementations operate on 64 bits directly, avoiding state encoding and decoding (which happens at least once for every game state in the course of computing a strong solution).

$$\begin{bmatrix}[\text{“x”}, \text{“o”}, \text{“-”}],\\ [\text{“o”}, \text{“x”}, \text{“-”}],\\ [\text{“x”}, \text{“o”}, \text{“-”}]\end{bmatrix} \longleftrightarrow \text{0b000...00101011110110} = 1398 \in \mathbb{N}_{2^{64}}$$

Computation

We can now look at algorithms that consume this interface. Consider the following implementation of Minimax as a representative example:

- Perform a full exploration of game states.
- Keep track of the utility of each player at terminal states, and mark them as solved.
- For each **parent state** of a solved state, set its utility vector to that of the child which would maximize the utility of the player whose turn it is at the parent state, and mark it as solved.
- Repeat the previous step until all states are marked as solved.

Obtain parent state through the construction of an in-memory graph data structure in $O(|S|^2)$ space, or add a "retrograde" function to the Game interface to keep space usage at a minimal $O(|S|)$.

Note: The algorithm outline above is not correct for cyclic games.

Computation

We can now look at algorithms that consume this interface. Consider the following implementation of Minimax as a representative example:

- Perform a full exploration of game states.
- Keep track of the utility of each player at terminal states, and mark them as solved.
- For each **parent state** of a solved state, set its utility vector to that of the child which would maximize the utility of the player whose turn it is at the parent state, and mark it as solved.
- Repeat the previous step until all states are marked as solved.

Obtain parent state through the construction of an in-memory graph data structure in $O(|S|^2)$ space, or add a "retrograde" function to the Game interface to keep space usage at a minimal $O(|S|)$.

Note: The algorithm outline above is not correct for cyclic games.

Computation

We can now look at algorithms that consume this interface. Consider the following implementation of Minimax as a representative example:

- Perform a full exploration of game states.
- Keep track of the utility of each player at terminal states, and mark them as solved.
- For each **parent state** of a solved state, set its utility vector to that of the child which would maximize the utility of the player whose turn it is at the parent state, and mark it as solved.
- Repeat the previous step until all states are marked as solved.

Obtain parent state through the construction of an in-memory graph data structure in $O(|S|^2)$ space, or add a "retrograde" function to the Game interface to keep space usage at a minimal $O(|S|)$.

Note: The algorithm outline above is not correct for cyclic games.

Computation

We can now look at algorithms that consume this interface. Consider the following implementation of Minimax as a representative example:

- Perform a full exploration of game states.
- Keep track of the utility of each player at terminal states, and mark them as solved.
- For each **parent state** of a solved state, set its utility vector to that of the child which would maximize the utility of the player whose turn it is at the parent state, and mark it as solved.
- Repeat the previous step until all states are marked as solved.

Obtain parent state through the construction of an in-memory graph data structure in $O(|S|^2)$ space, or add a "retrograde" function to the Game interface to keep space usage at a minimal $O(|S|)$.

Note: The algorithm outline above is not correct for cyclic games.

Computation

We can now look at algorithms that consume this interface. Consider the following implementation of Minimax as a representative example:

- Perform a full exploration of game states.
- Keep track of the utility of each player at terminal states, and mark them as solved.
- For each **parent state** of a solved state, set its utility vector to that of the child which would maximize the utility of the player whose turn it is at the parent state, and mark it as solved.
- Repeat the previous step until all states are marked as solved.

Obtain parent state through the construction of an in-memory graph data structure in $O(|S|^2)$ space, or add a "retrograde" function to the Game interface to keep space usage at a minimal $O(|S|)$.

Note: The algorithm outline above is not correct for cyclic games.

Computation

We can now look at algorithms that consume this interface. Consider the following implementation of Minimax as a representative example:

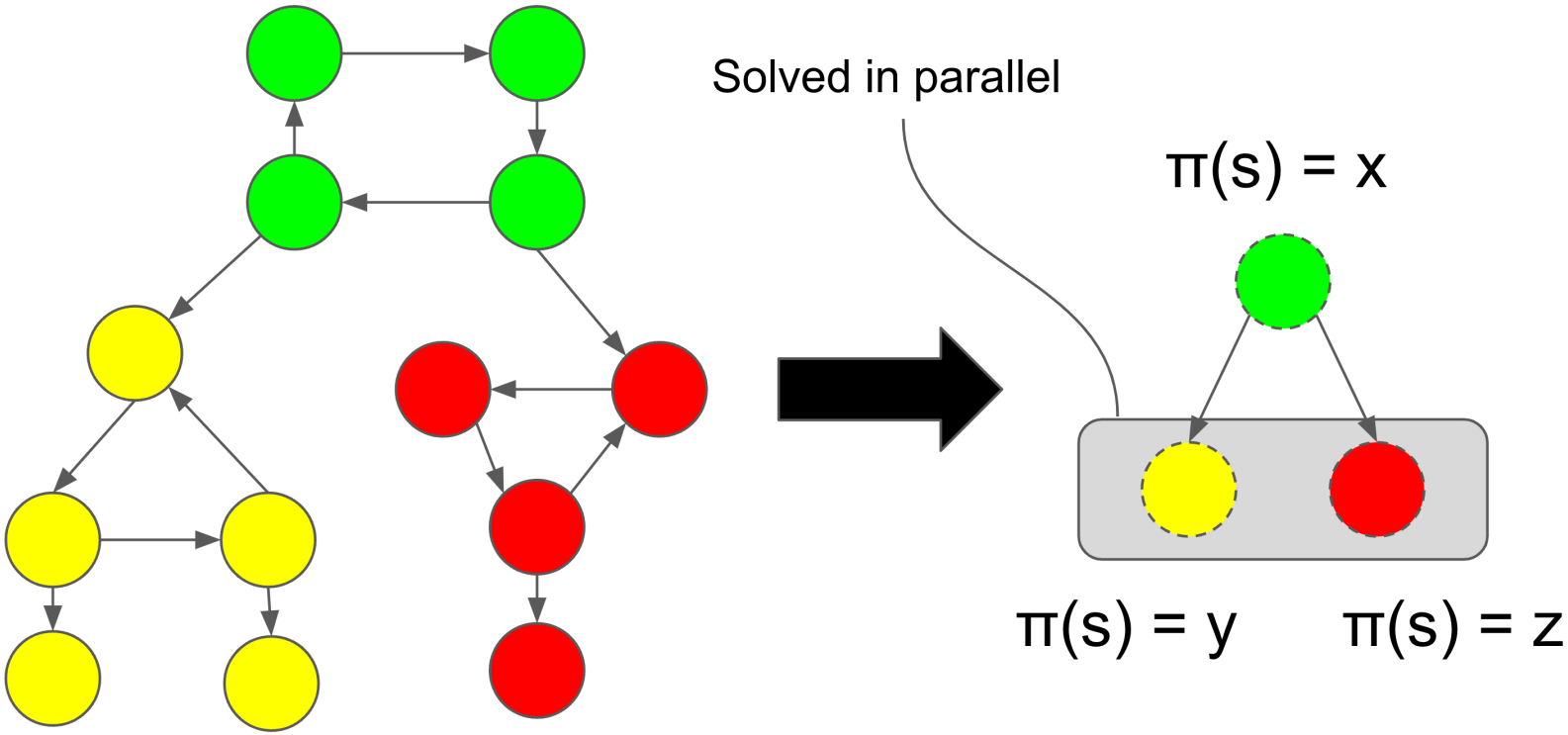
- Perform a full exploration of game states.
- Keep track of the utility of each player at terminal states, and mark them as solved.
- For each **parent state** of a solved state, set its utility vector to that of the child which would maximize the utility of the player whose turn it is at the parent state, and mark it as solved.
- Repeat the previous step until all states are marked as solved.

Obtain parent state through the construction of an in-memory graph data structure in $O(|S|^2)$ space, or add a "retrograde" function to the Game interface to keep space usage at a minimal $O(|S|)$.

Note: The algorithm outline above is not correct for cyclic games.

Computation

In a general case, the time complexity of search is linear in the size of the game graph: $O(|S|^2)$. In certain cases, it is worthwhile to parallelize the process by defining a "partition" function $\pi : S \to \mathbb{N}$ in a way that minimizes the sum of the conductance of all partition cuts.



Storage

Solving algorithms generate streams of output (utility and other attributes bound to states) that need to be efficiently stored and retrieved:

- Memory is rarely enough to keep all the information we care about.
- It is orders of magnitude slower to perform disk operations.
- Custom database systems minimize the number of disk operations needed throughout the execution of a solving algorithm.
- There are countless considerations in database systems...

Storage

Solving algorithms generate streams of output (utility and other attributes bound to states) that need to be efficiently stored and retrieved:

- Memory is rarely enough to keep all the information we care about.
- It is orders of magnitude slower to perform disk operations.
- Custom database systems minimize the number of disk operations needed throughout the execution of a solving algorithm.
- There are countless considerations in database systems...

Storage

Solving algorithms generate streams of output (utility and other attributes bound to states) that need to be efficiently stored and retrieved:

- Memory is rarely enough to keep all the information we care about.
- It is orders of magnitude slower to perform disk operations.
- Custom database systems minimize the number of disk operations needed throughout the execution of a solving algorithm.
- There are countless considerations in database systems...

Storage

Solving algorithms generate streams of output (utility and other attributes bound to states) that need to be efficiently stored and retrieved:

- Memory is rarely enough to keep all the information we care about.
- It is orders of magnitude slower to perform disk operations.
- Custom database systems minimize the number of disk operations needed throughout the execution of a solving algorithm.
- There are countless considerations in database systems...

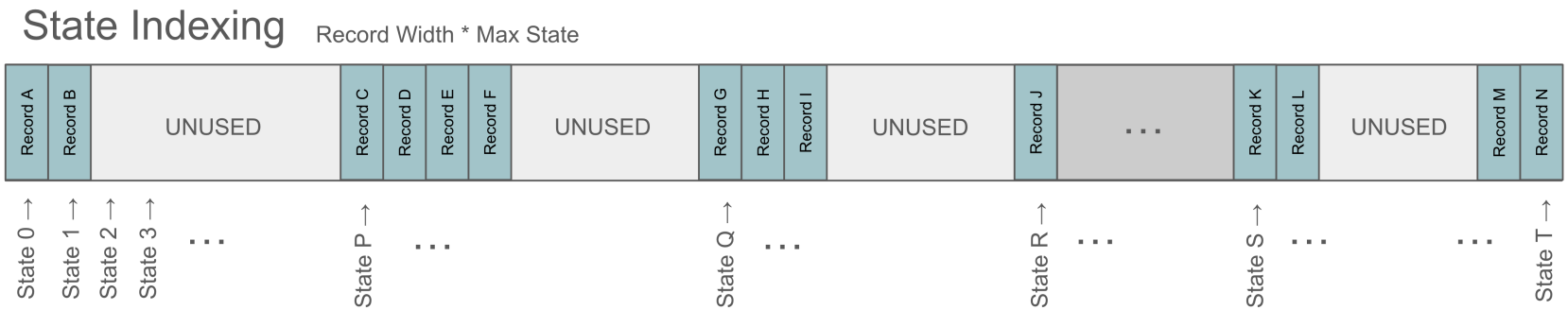
Storage

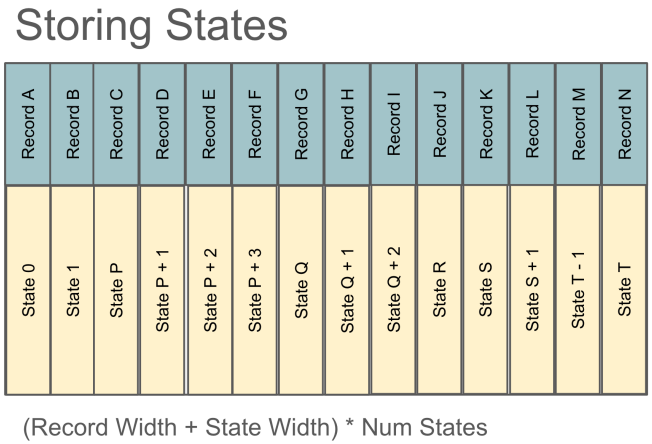
Solving algorithms generate streams of output (utility and other attributes bound to states) that need to be efficiently stored and retrieved:

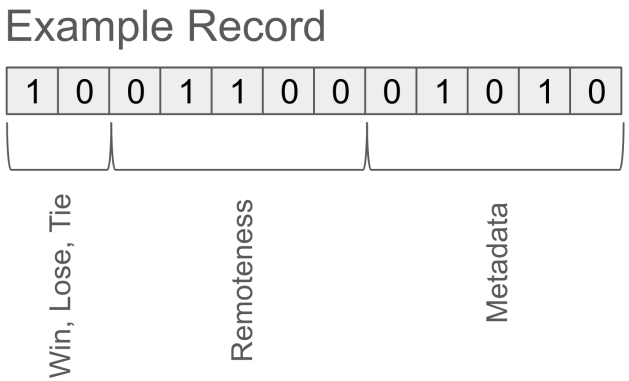
- Memory is rarely enough to keep all the information we care about.
- It is orders of magnitude slower to perform disk operations.
- Custom database systems minimize the number of disk operations needed throughout the execution of a solving algorithm.
- There are countless considerations in database systems...

Storage

Example tradeoff: Storing states or using them as indices, featuring sparse hash functions.







Thank you

In summary, we have gone through how we **represent** games, **compute** algorithms that consume them, and **store** generated information in a way that allows us to **extend the depth of search** in game-theoretic systems.

Our group's work: https://github.com/GamesCrafters

Our strong solutions: https://nyc.cs.berkeley.edu/uni/about

Lingering questions: maxfierro@berkeley.edu

References


A. R. Martin J. Osborne, *A Course In Game Theory*. Cambridge: The MIT Press, 1994.


T. Sandholm, "Abstraction for solving large incomplete-information games," *Association for the Advancement of Artificial Intelligence*, 2015.