A few things set two-player games apart from standard search problems:

- Opponents — the next stage is unpredictable, so we need a strategy
- Time Limits — search time is limited, so a good, fast result is better than a perfect slow one

Formulating 2-Player Games As Search

Consider Chess; we have two major issues to address:
The effective branching factor is roughly 35, so the tree is huge.
We don't know the moves the opponent will make.

What we need is a **conditional plan** (equivalent to a strategy).
This solution will thus include:

1. a goal state at every leaf
2. an action at each of our turns
3. a path for each of the foe's choice

Our approach:

![]($(https$ $:$ $//paper$ $-$
$attachments.dropbox.com/s_{B}F954AA1A932522D387644125A680A937DC67A6EB823204801E178C7E92B0DE9_{1}588283930207_{U}$
$drawing + 7.jpg)$

We only need an algorithm which chooses from A1, A2, A3, since we can reapply each turn.
We assume that the opponent is perfect, since if they aren't we will only do better.
If choose A_1 — then our opponent will choose 3, thus we want to choose the best worst case.

We consider this approach bottom-up, since we compute scores starting at the bottom.
How do we build the tree? DFS.
Recall space = O(bm) & time = O(b^m)

Consider chess: it has a branching factor of roughly 35.
The time limit is often 100 seconds.
We can generate 10^4 nodes/second.
$\implies$ choices = $log_35(10^6)$ = 4, so our approach is 2 PLY. Is that good?
  4-ply look ahead — human novice
  8-ply look ahead — human master
  12-ply look ahead — Kasparov, Deep Blue
Thus our method is not nearly sufficient.

We need to "cut m" via an **evaluation function**
  $\equiv$ a function which attempts to approximate the minimal node value.
Requirements for the function:

1. agrees with the utility function on terminal nodes
2. is efficient to compute
3. is accurate in premise

Note that this may lead us to cut off paths at different steps.

Using the evaluation function, we can skip any value that is lower than the min encountered!
This is called **alpha-beta pruning** (alpha for max, beta for minimum).

![]$(https$ $:$ $//paper$ $-$
$attachments.dropbox.com/s_BF954AA1A932522D387644125A680A937DC67A6EB823204801E178C7E92B0DE9_1588285154036$
$drawing + 8.jpg)$

Our choice is dependent on the type of node currently being examined:
min node — examine children lowest-first
max node — examine children largest-first

The gain for alpha-beta pruning depends on value order.
$\Longrightarrow$ with a-b pruning, $T = O(b^{m/2})$, or we can search twice as deep.
This is the best case, the average is $O(b^{3m/4})$.

Stochastic Search

This strategy cannot work for games of chance since we cannot predict our choices.
We can instead use probabilities to weight the values.

![$Notethatourchoicesaremaintainedforlineartransformations.$
$](https$ $:$ $//paper$ $-$
$attachments.dropbox.com/s_BF954AA1A932522D387644125A680A937DC67A6EB823204801E178C7E92B0DE9_1588286875004_l$
$drawing + 9.jpg)$

Our choice of cutoff is very important:
A depth cutoff in a search (here 2) must be **quiescent** ≡ unlikely to drastically change values.
A bad cutoff may fall victim to the **horizon effect** ≡ when negatives are pushed just past the cutoff.
We address this by logging clearly great moves for later.
Time complexity: $O(b^m n^m)$ for n distinct stochastic events.

Additional Game Strategies

At the beginning or the end of a game, it may be more practical to do a table lookup.
≡ create a policy (a mapping from every single state to its optimal reachable state).
We can then do a **retrograde** (≡ bottom up) search from an ideal to choose our action.

Probabilistic & Perfect Information

1. define a chance node which holds the expected value of its children
2. **Monte Carlo Simulation** ≡ estimate values by win percentage

Probabilistic & Imperfect Information

1. **averaging over clairvoyance** ≡ solving over all possible "rolls"
   This has the negative tendency to avoid moves that gain the player information
2. introduce randomness for an equilibrium solution