Game-Tree Search over High-Level Game States in RTS Games, by A. Uriarte and S. Ontañón

Marko Obradovic

University of Basel

Search and Optimizatin Seminar

November 5, 2015

Outline

1. Introduction
   - Characteristics of Real-Time Strategy Games
   - Motivation
2. High-level Abstraction in RTS Games
   - State Representation
   - Actions
3. High-Level Game-Tree Search
   - High-Level State Forwarding
   - High-Level State Evaluation
   - High-Level Game-Tree Search Algorithms
4. Experimental Results
5. Conclusion

Characteristics of Real-Time Strategy Games

Real-Time Strategy Games

In *Real-Time Strategy* (RTS) games the player's goal is to gather resources, build bases and establish military power in order to win against his opponents.

RTS games have the following important characteristics:

- *simutaneous moves*
- *durative actions*
- *real-time game playing*

e.g. *StarCraft* where players participate in a wage war across the galaxy

Motivation

Problems of RTS games

Due to their characteristics, RTS games have an **enormous state space** which results in a **very large branching factor**. Thus, the applicability of game-tree search algorithms is very limited.

Given a *game state*, there are...

- many possible actions a player could execute
- many possible paths in a game-tree

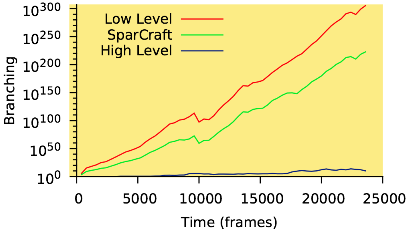


Figure: e.g. branching factor in StarCraft

Motivation

Basic Idea $\hat{=}$ Abstraction

Address large branching factors by introducing a **high-level game state representation**.

Goals of high-level game state representation:

- significantly **shrink state space**
  $\rightarrow$ *enable applicability of game-tree search algorithms*

Focus on *combat scenarios*:

- capture only army information
  $\rightarrow$ *placement of combat units in the game state*
- apply game-tree search for controlling high-level army movements
  $\rightarrow$ *actions for combat units*

High-level Abstraction in RTS Games

The high-level game state abstraction involves two important elements:

State Representation

Basic idea:

- **decompose map into regions** by grouping unblocked cells
  → *ignore single cells*
- **group combat units** by region and unit type
  → *ignore individual units and their exact (x,y)-location on the grid*

Actions

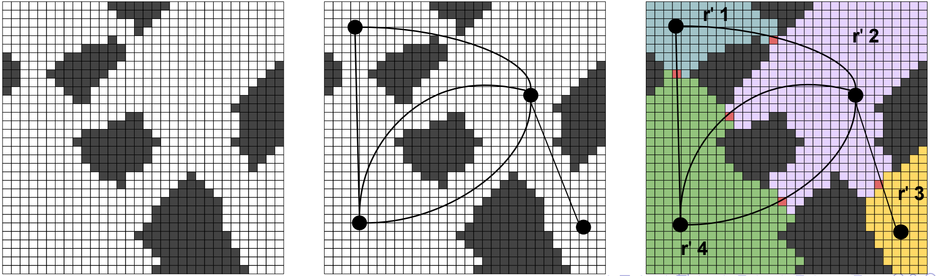
Basic idea:

- define a set of **high-level actions** that can be applied to groups of combat units
  → *ignore actions of single combat units*

State Representation

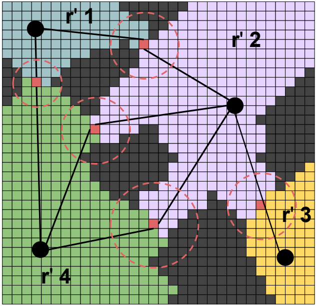
Map decomposition: Step #1

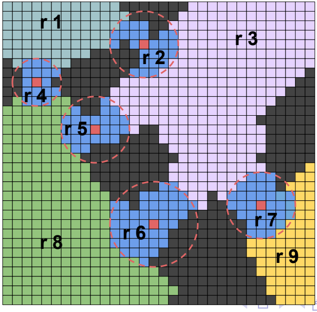
- Perkins algorithm: divide map into set of **regions** $R' = \{r'_1, ..., r'_m\}$ connected by **chokepoints** $C = \{c_1, ..., c_q\}$
- Chokepoints: cells (red) that define strategic bottlenecks
  → *connectors of exactly two regions*
- As graph: regions r'_i are vertices, chokepoints c_j are edges (later also vertices)



Map decomposition: Step #2

- Chokepoint cells are centers of bottleneck regions
- Margin of the bottleneck region defined by **radius** of chokepoint cell
- Cells **within** margin belong to the region generated from chokepoint
- Remaining regions contain cells from r_i' that are **not** part of any region around a chokepoint



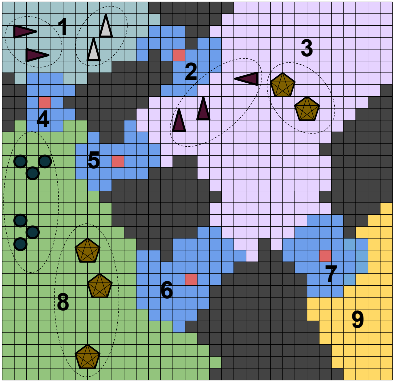


Result:
set of regions
$R = \{r_1, \ldots, r_n\}$

State Representation

Grouping of combat units

- Combat units are grouped **by unit type** and **by region**
  → *if same unit type and located in same region, then same combat unit group, e.g.* $Tank_{Region3}$



Unit Type	Size	Region
Tank	3	3
Tank	2	1
Tank	2	1
Base	2	3
Base	3	8
Marine	6	8

Actions

Set of high-level actions

- **N/A**: action only for army buildings (e.g. bases) and means *do nothing* since buildings cannot perform any combat action
- **Move**: move from current region to a neighboring region
- **Attack**: attack any enemy in current region
- **Idle**: do nothing during 400 game frames

Note: High-level actions are applied to whole combat unit groups, not to individuals

State Representation and Actions: Example

Exemplary high-level game state representation table resulting from previous map:

Player	Unit Type	Size	Region	Order	Target	End Frame
1	Tank	3	3	Move	2	240
1	Tank	2	1	Attack	1	370
2	Tank	2	1	Idle	-	400
1	Base	2	3	N/A	-	-
1	Base	3	8	N/A	-	-
1	Marine	6	8	Move	4	150

Table: Units grouped by region and type

High-Level Game-Tree Search

High-level game-tree search requires two more pieces:

High-Level State Forwarding

- RTS games have **simultaneous** and **durative** actions
- During game-tree search the game is not played
  $\rightarrow$ *no waiting for action to complete*
- Idea: jump to next **decision point** where at least one player can execute an action

High-Level State Evaluation

- Game-tree search algorithms need to choose a game state node to expand according to **adequate criteria**
- Idea: define **evaluation function** that tells how **good** a high-level game state is

High-Level State Forwarding

High-level state forwarding has two components:

End frame prediction

- Jumping to next decision point requires knowing the **end frames** of actions
  → *estimate the duration of actions, i.e. when they are completed*
- e.g. Move action: end frame estimation based on **distances** between region centers and **speed** of combat unit types

Simulation

- Identification of **smallest end frame** of a game state and jump there
- After jumping **outcome update** of completed actions required
- e.g. Move action: update group position with target position

High-Level State Evaluation

- Game state evaluation functions define how good a game state is
- Example in paper: Use **destroy score** of a combat unit in StarCraft
  $\rightarrow$ *consider the costs (e.g. for the resources) to build that unit*

High-Level Game-Tree Search Algorithms

Two high-level game-tree search algorithms were used for the experiments:

- **Monte Carlo Tree Search Considering Durations (MCTSCD)**
- **Alpha-Beta Considering Durations (ABCD)**

Note: both are extensions of standard tree search algorithms to deal with simultaneous moves and durative actions

Experimental Results

Evaluation using two StarCraft game maps:

ABCD and MCTSCD tested against scripted algorithm and built-in AI

Win ratio comparison

hand-scripted (100%) $>$ MCTSD ($\sim$90%) $>$ ABCD ($\sim$80%) $>$ StarCraft's built-in AI ($\sim$20%)

Branching factor

only grows beyond 10^{10} when large number of groups ($\geqslant 30$)

Conclusion

Presented game state abstraction is **useful**:

- branching factor highly reduced
- state space shrinked and simplified, but still between 80% and 90% of wins achieved in experiments
- StarCraft's built-in AI defeated

$\rightarrow$ problem transformed to a level that can be handled by game-tree search
$\rightarrow$ resulting actions are meaningful in the game

Weak points:

- some facts are not further explained (e.g. search every 400 frames)
- game paused while search is performed
- approach limited to combat scenarios

Questions?