Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment

Build Order Optimization in StarCraft

David Churchill and Michael Buro
University of Alberta
Computing Science Department
Edmonton, Alberta, Canada

Abstract

In recent years, real-time strategy (RTS) games have gained interest in the AI research community for their multitude of challenging subproblems — such as collaborative pathfinding, effective resource allocation and unit targeting, to name a few. In this paper we consider the build order problem in RTS games in which we need to find concurrent action sequences that, constrained by unit dependencies and resource availability, create a certain number of units and structures in the shortest possible time span. We present abstractions and heuristics that speed up the search for approximative solutions considerably in the game of StarCraft, and show the efficacy of our method by comparing its real-time performance with that of professional StarCraft players.

Introduction

Automated planning, i.e. finding a sequence of actions leading from a start to a goal state, is a central problem in artificial intelligence research with many real-world applications. For instance, the satisfiability problem can be considered a planning problem (we need to assign values to n Boolean variables so that a given formula evaluates to true) with applications to circuit verification, solving Rubik's cube from a given start state is a challenging pastime, and building submarines when done inefficiently can easily squander hundreds of millions of dollars. Most interesting planning problems in general are hard to solve algorithmically. Some, like the halting problem for Turing machines, are even undecidable.

In this paper we consider a class of planning problems that arises in a popular video game genre called real-time strategy (RTS) games. In these games, which can be succinctly described as war simulations, players instruct units in real-time to gather resources, to build other units and structures, to scout enemy locations, and to eventually destroy opponents' units to win the game. In the opening phase of RTS games players usually don't interact with each other because their starting locations are spread over large maps and player visibility is limited to small regions around friendly units or structures. The main sub-goals in this game phase are to establish a sufficient income flow by producing workers that gather resources, to quickly build structures that are prerequisites for other structures or can produce combat units to build a minimal force for defense or early attack, and to send out scouts to explore the terrain and search for enemy bases. The order in which units and structures are produced is called a *build order*. RTS games are usually won by players who destroy opponents' structures first. This goal can be accomplished in various ways. For instance, one could try to surprise ("rush") the opponent by investing resources into attack forces early in the game at the cost of delaying the construction of structures that are important in later game stages. If the opponent, on the other hand, invests in technological development and disregards defense, the rushing player will win easily. Thus, at the highest adversarial strategy level, the choice of initial build orders often decides the game outcome. Therefore, like in chess, aspiring players need to study and practice executing build orders and tailor them to specific opponents. Avoiding the interesting and ambitious task of selecting good build order goals, in this paper we assume that they are given to us. Because acting fast is very important in RTS games due to the fact that players act asynchronously, what remains is finding action sequences that accomplish the given goal while minimizing the plan duration (makespan). This process is called *build order optimization*.

The research on this subject that is reported here was motivated by the goal of creating strong AI systems for the popular RTS game of StarCraft and the frustration of hard-coding build orders for them. In the remainder of this paper we first give a brief overview of related work on build order optimization and our application domain StarCraft. Then we describe our search algorithm, the underlying assumptions, and the abstractions we use. In the following experimental section we gauge the performance of our planner by comparing its build orders with those executed by professional players. We finish the paper by conclusions and suggestions for future work.

Background

The build order optimization problem can be described as a constraint resource allocation problem with makespan minimization, which features concurrent actions. Because of their practical relevance, problems of this kind have been the subject of study for many years, predominantly in the area of operations research.

(Buro and Kovarsky 2007) motivates research on build order problems in the context of RTS games and proposes a

way of modeling them in PDDL, the language used in the automated planning competitions. In (Kovarsky and Buro 2006) the issue of concurrent execution is studied in general and efficient action ordering mechanisms are described for the RTS game build order domain.

Existing techniques for build order planning in the RTS game domain have focused mainly on the game Wargus (an open source clone of Warcraft 2), which is much simpler than StarCraft due to the limited number of possible actions and lower resource gathering complexity. Several of these techniques rely heavily on means-end analysis (MEA) scheduling. Given an initial state and a goal, MEA produces a satisficing plan which is minimal in the number of actions taken. MEA runs in linear time w.r.t. the number of actions in a plan, so it is quite fast, but the makespans it produces are often much longer than optimal.

(Chan et al. 2007b) employ MEA to generate build order plans, followed by a heuristic rescheduling phase which attempts to shorten the overall makespan. While they produce satisficing plans quite quickly, the plans are not optimal due to the complex nature of the rescheduling problem. In some cases they are able to beat makespans generated by human players, but do not mention the relative skill level of these players. This technique is extended in (Chan et al. 2007a) by incorporating best-first search in an attempt to reduce makespans further by solving intermediate goals. They admit that their search algorithm is lacking many optimizations, and their results show that this is not only slower than their previous work but still cannot produce significantly better solutions. (Branquinho and Lopes 2010) extend further on these ideas by combining two new techniques called MeaPop (MEA with partial order planning) and Search and Learning A* (SLA*). These new results improve on the makespans generated by MEA, but require much more time to compute, bringing it outside the range of real-time search. They are currently investigating ways of improving the run-time of SLA*.

These techniques however are only being applied to Wargus, with goals consisting of at most 5 types of resources. Interesting plans in StarCraft may involve multiple instances of up to 15 different units in a single goal and requiring far more workers, increasing complexity dramatically.

StarCraft

RTS games are interesting application domains for AI researchers, because state spaces are huge, actions are concurrent, and part of the game state is hidden from players — and yet, human players still play much better than machines. To spark researchers' interest in this game domain, a series of RTS game AI competitions have been organized in the past 6 years. In 2006-2009 a free software RTS game engine was used (ORTS 2010), but since the advent of the BWAPI library (BWAPI 2011), the competition focus has switched to StarCraft (by Blizzard Entertainment), the most popular RTS game in the world with over 10 million copies sold. StarCraft has received over 50 game industry awards, including over 20 "game of the year" awards. Some professional players have reached celebrity status, and prize money for tournaments total in the millions of dollars annually.

As in most RTS games, each player starts with a number of worker units which gather resources such as minerals and gas which are consumed by the player throughout the game. Producing additional worker units early in the game increases resource income and is typically how most professional players start their build orders. Once a suitable level of income has been reached, players begin the construction of additional structures and units which grow their military strength. Each unit has a set of prerequisite resources and units which the player must obtain before beginning their construction. The graph obtained by listing all unit prerequisites and eliminating transitive edges is called a *tech tree*. Due to the complex balance of resource collection and unit prerequisite construction, finding good build orders is a difficult task, a skill often taking professional players years to develop.

A Build Order Planning Model for StarCraft

Build order planning in RTS games is concerned with finding a sequence of actions which satisfies a goal with the shortest makespan. It is our goal to use domain specific knowledge to limit both the branching factor as well as depth of search while maintaining optimality, resulting in a search algorithm which can run in real-time in a StarCraft playing agent.

In StarCraft, a player is limited to a finite number of resources which they must both collect and produce throughout the game. All consumables (minerals, gas) as well as units (workers, fighters, buildings) are considered resources for the purpose of search. An action in our search is one which requires some type of resource, while producing another (combat actions are out of our scope). Resources which are used by actions can be of the forms Require, Borrow, Consume, and Produce (Branquinho and Lopes 2010). Required resources, which are called prerequisites, are the ones which must be present at the time of issuing an action. A borrowed resource is one which is required, used for the duration of an action, and returned once the action is completed. A consumed resource is one which is required, and used up immediately upon issue. A produced resource is one which is created upon completion of the action.

Each action a has the form $a = (\delta, r, b, c, p)$, with duration δ (measured in game simulation frames), three sets of preconditions r (required), b (borrowed), c (consumed), and one set of produced items p. For example, in the StarCraft domain, the action $a =$ "Produce Protoss Dragoon" has $\delta = 600$, $r = \{\text{Cybernetics-Core}\}$, $b = \{\text{Gateway}\}$, $c = \{125$ minerals, 50 gas, 2 supply$\}$, $p = \{1$ Dragoon$\}$.

States then take the form $S = (t, R, P, I)$, where t is the current game time (measured in frames), vector R holds the state of each resource available (ex: 2 barracks available, one currently borrowed until time X), vector P holds actions in progress but are not yet completed (ex: supply depot will finish at time X), and vector I holds worker income data (ex: 8 gathering minerals, 3 gathering gas). Unlike some implementations such as (Branquinho and Lopes 2010), I is necessary due to abstractions made to facilitate search.

Abstractions

Without having access to the StarCraft game engine source code, it was necessary to write a simulator to compute

state transitions. Several abstractions were made in order to greatly reduce the complexity of the simulation and the search space, while maintaining close to StarCraft-optimal results. Note that any future use of the term 'optimal' or 'optimality' refers to optimality within these abstractions:

We abstract mineral and gas resource gathering by real valued income rates of 0.045 minerals per worker per frame and 0.07 gas per worker per frame. These values have been determined empirically by analyzing professional games. In reality, resource gathering is a process in which workers spend a set amount of time gathering resources before returning them to a base. Although we fixed income rates in our experiments, they could be easily estimated during the game. This abstraction greatly increases the speed of state transition and resource look-ahead calculations. It also eliminates the need for "gather resource" type actions which typically dominate the complexity of build order optimization. Due to this abstraction, we now consider minerals and gas to be a special type of resource, whose "income level" data is stored in state component I.

Once a refinery location has been built, a set number of workers (3 in our experiments) will be sent to gather gas from it. This abstraction eliminates the need for worker reassignment and greatly reduces search space, but in rare cases is not "truly" optimal for a given goal.

Whenever a building is constructed, a constant of 4 seconds (96 simulation frames) is added to the game state's time component. This is to simulate the time required for a worker unit to move to a suitable building location within an arbitrary environment, since individual map data is not used in our search, but again could be estimated during the game.

Algorithm

We use a depth-first branch and bound algorithm to perform build order search. The algorithm takes a starting state S as input and performs a depth-first recursive search on the descendants of S in order to find a state which satisfies a given goal G. This algorithm has the advantage of using a linear amount of memory with respect to the maximum search depth. Since this is an any-time algorithm we can halt the search at any point and return the best solution so far, which is an important feature for real-time applications.

Action Legality

In order to generate the children of a state, we must determine which actions are legal in this state. Intuitively, an action is legal in state S if the simulation of the game starting in time will eventually produce all required resources without issuing any further actions. Given our abstractions, an action is therefore legal in state S if and only if the following conditions hold: 1) The prerequisites required or resources borrowed are either currently available, or being created. Example: a Barracks is under construction, so fighter units will be trainable without any other actions being issued. 2) The consumed resources required by the action are either currently available or will be available at some point in the future without any other actions being taken. Example: we do not currently meet the amount of minerals required, however our workers will eventually gather the required amount (assuming there is a worker gathering minerals).

Algorithm 1 Depth-First Branch & Bound

Require: goal G, state S, time limit t, bound b

```
1:  procedure DFBB(S)
2:      if TimeElapsed ≥ t then
3:          return
4:      end if
5:      if S satisfies G then
6:          b ← min(b, S_t)                          ▷ update bound
7:          bestSolution ← solutionPath(S)
8:      else
9:          while S has more children do
10:             S' ← S.nextChild
11:             S'.parent ← S
12:             h ← eval(S')                         ▷ heuristic evaluation
13:             if S'_t + h < b then
14:                 DFBB(S')
15:             end if
16:         end while
17:     end if
18: end procedure
```

Fast Forwarding and State Transition

In general, RTS games allow the user to take no action at any given state, resulting in a new state which increases the internal game clock, possibly increasing resources and completing actions. This is problematic for efficient search algorithms since it means that all actions (including the null action) must be taken into consideration in each state of the game. This results in a search depth which is linear not in the number of actions taken, but in the makespan of our solution, which is often quite high. In order to solve this problem, we have implemented a *fast-forwarding* simulation technique which eliminates the need for null actions.

In StarCraft, the time-optimal build order for any goal is one in which actions are executed as soon as they are legal, since hoarding resources cannot reduce the total makespan. Although resource hoarding can be a vital strategy in late-game combat, it is outside the scope of our planner. Let us define the following functions:

$S' \leftarrow \text{Sim}(S, \delta)$ - Simulate the natural progression of a StarCraft game from a state S through δ time steps given that no other actions are issued, resulting in a new state S'. This simulation includes the gathering of resources (given our economic abstraction) and the completion of durative actions which have already been issued.

$\delta \leftarrow \text{When}(S, R)$ - Takes a state S and a set of resource requirements R and returns the earliest time δ for which $\text{Sim}(S, \delta)$ will contain R. This function is typically called with action prerequisites to determine when the required resources for an action a will be ready.

$S' \leftarrow \text{Do}(S, a)$ - Issue action a in state S assuming all required resources are available. The issuing of the action involves subtracting the consume resources, updating actions in progress and flagging borrowed resources in use. The resulting state S' is the state for which action a has just been

issued and has its full duration remaining.

$$S' = \text{Do}(\text{Sim}(S, \text{When}(S,a)),a)$$

now defines our state transition function which returns the state S' for which action a has been issued.

Concurrent Actions and Action Subset Selection

A defining feature of RTS games is the ability to perform concurrent actions. For example, if a player has a sufficient amount of resources they may begin the concurrent construction of several buildings as well as the training of several units. In a general setting, this may cause an action-space explosion because an super-exponential number of possible actions sequences has to be considered. Even in the common video game setting in which a game server sequentializes incoming concurrent player actions, it can be co-NP hard to decide whether these actions when sequentialized in arbitrary order result in the same state (Buro and Kovarsky 2007). Fortunately, many RTS games, including StarCraft, have the property that simultaneously executable actions are independent of each other, i.e. action effects don't invalidate prerequisites of other actions: For any two actions a,b to be executable concurrently in state S we must have $\delta = \text{When}(S, \text{prerequisites of } a \text{ and } b) = 0$, which means $\text{Sim}(S,\delta) = S$. Because function $\text{Do}(S,x)$ returns a state in which precondition resources are decreased and postcondition resources are increased, we have

$$\begin{aligned} \text{Do}(\text{Do}(S,a),b) &= \text{Do}(S,a+b) \\ &= \text{Do}(\text{Do}(S,b),a), \end{aligned}$$

where '+' indicates the concurrent issuing of two actions, proving that the ordering of concurrent actions has no effect on the resulting state. We can also apply this argument iteratively for subsets larger than two actions. Based on this insight and the "earliest execution" property of optimal action sequences we discussed in the previous subsection, we can therefore impose a single ordering on simultaneous actions to to eliminate the need for iterating over all possible sequences of concurrent actions from a given state.

Heuristics

Our depth-first branch and bound algorithm allows us to prune nodes based on heuristic evaluations of the path length left to our goal. Line 13 of Algorithm 1 shows that we can prune a child node if its length so far plus its heuristic evaluation is less than the upper bound. If our heuristic is admissible, this guarantees that our computed solution will be optimal. We use the following admissible lower-bound heuristics to prune our search:

- LandmarkLowerBound(*S*,*G*) — StarCraft's tech tree imposes many prerequisites on actions. These actions are known in the search literature as landmarks. Given this sequence of non-concurrent landmark actions, we sum the individual durations of actions not yet created to form an admissible lower bound for our search.

- ResourceGoalBound(*S*,*G*) — Summing the total consumed resource cost of units in a goal gives us a lower bound on the resources required to construct the goal optimally. Performing a quick search to determine the makespan of producing only these resources is an admissible heuristic.

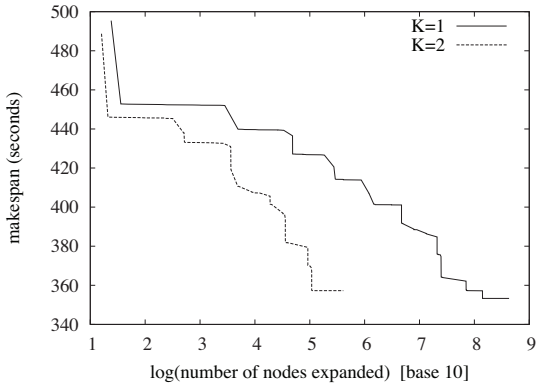


Figure 1: Makespan vs. nodes searched for late-game goal of two carriers, comparing optimal search ($K = 1$) and approximate search with macro actions ($K = 2$). Macro actions make complex searches tractable while maintaining close to optimal makespans.

We can then take the maximum of these three heuristics as our heuristic value *h*. The heuristic used as an upper bound for our search is TrivialPlan(*S*,*G*) — Given a state and a goal, we simply take a random legal action from the goal and issue it when it is possible. This guarantees that our goal is met, but does not optimize for time. The length of this plan is then used as an upper bound in our search.

Breadth Limiting

To limit the branching factor of our search, we impose upper bounds on certain actions. Ex: if our goal contains two fighter units which are trained at a barracks, we know that we need to produce at most two barracks. Since it is difficult to pre-compute the optimal number of worker and supply units for a given goal in this fashion, higher bounds are placed on them to ensure optimal numbers can be produced.

Macro Actions

Macro actions (also called *options* in reinforcement learning) have proven useful in speeding up search and planning through incorporating domain specific knowledge (Iba 1989). While these actions can be learned (Stolle and Precup 2002), we have simply hand-created several macro actions by inspecting build orders used by professional players. Our macros all take the form of *doubling* existing actions which are commonly executed in sequence. For example: professional players often build worker or fighter units in bunches, rather than one at a time. By creating macro actions such as these we cut the depth of search dramatically while maintaining close to time-optimal makespans. To implement this, for each action we associate a repetition value *K* so that only *K* actions in a row of this type are allowed. The effects of introducing macro actions can be seen in Figure 1.

Experiments

Experiments were conducted to compare build orders used by professional StarCraft players to those produced by our planner. Although our planner is capable of planning for

Algorithm 2 Compare Build Order

Require: BuildOrder B, TimeLimit t, Increment Time i

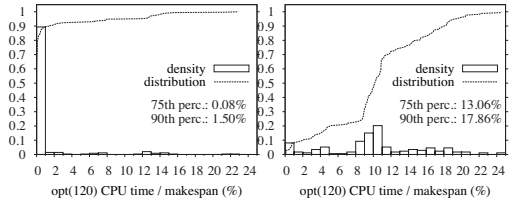
procedure COMPAREBUILDORDER(B,t,i)
  $S \leftarrow$ Initial StarCraft State
  SearchPlan $\leftarrow$ DFBB(S,GetGoal($B,0,\infty$),t)
  if SearchPlan.timeElapsed $\leq t$ **then**
    return MakeSpan(SearchPlan) / MakeSpan(B)
  else
    inc $\leftarrow i$
    SearchPlan $\leftarrow \emptyset$
    while inc $\leq$ MakeSpan(B) **do**
      IncPlan $\leftarrow$ DFBB(S,GetGoal(B,inc-i,inc),t)
      if IncPlan.timeElapsed $\geq t$ **then**
        return failure
      else
        SearchPlan.append(IncPlan)
        $S \leftarrow S$.execute(IncPlan)
        inc $\leftarrow$ inc + i
      end if
    end while
    return MakeSpan(SearchPlan) / MakeSpan(B)
  end if
end procedure

each race, we limited our tests to Protoss players in order to avoid any discrepancies caused by using build orders of different races. 100 replays were chosen from various repositories online, 35 of which feature professional players Bisu, Stork, Kal, and White-Ra. The remaining replays were taken from high level tournaments such as World Cyber Games.

The BWAPI StarCraft programming interface was used to analyze and extract the actions performed by the professional players. Every 500 frames (21s) the build order implemented by the player (from the start of the game) was extracted and written to a file. Build orders were continually extracted until either 10000 frames (7m) had passed, or until one of the player's units had died. A total of 520 unique build orders were extract this way. We would like to have used more data for further confidence, however the process of finding quality replays and manually extracting the data was quite time consuming. Though our planner is capable of planning from any state of the game, the beginning stages were chosen as it was too difficult to extract meaningful build orders from later points in the game due to the on-going combat. To extract goals from professional build orders, we construct a function GetGoal(B,t_s,t_e) which given a professional build order sequence B, a start time t_s and an end time t_e computes a goal which contains all resources produced by actions issued in B between t_s and t_e.

Tests were performed on each build order with the method described in Algorithm 2 with both optimal (opt) and macro action (app) search. First with $t = 60s$ and $i = 15s$, second with $t = 120s$ and $i = 30s$. This incremental tactic is believed to be similar in nature to how professionals re-plan at various stages of play, however it is impossible be certain without access to professionally labeled data sets (for which none exist). We claim that build orders produced by this system are "real-time" or "online" since they consume

A) CPU time statistics for search without macro actions:



B) CPU time statistics for search with macro actions:

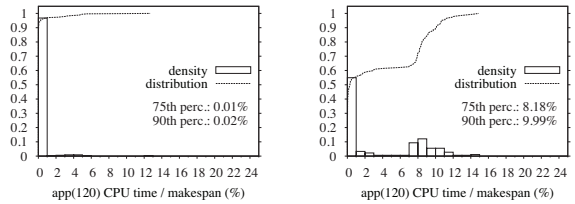


Figure 2: CPU time statistics for search without (A), and with (B) macro actions at 120s increments. Shown are densities and cumulative distributions of CPU time/makespan ratios in % and percentiles for professional game data points with player makespans 0..249s (left) and 250..500s (right). E.g. the top-left graph indicates that 90% of the time, the runtime is only 1.5% of the makespan, i.e. 98.5% of the CPU time in the early game can be used for other tasks.

far less CPU time than the durations of the makespans they produce. Agents can implement the current increment while it plans the next. It should be noted that this experiment is indeed biased against the professional player, since they may have changed their mind or re-planned at various stages of their build order. It is however the best possible comparison without having access to a professional player to implement build orders during the experiment.

Figures 2 (time statistics) and 3 (makespan statistics) display the results of these experiments, from which we can conclude our planner produces build orders with comparable makespans while consuming few CPU resources. Results for 60s incremental search were similar to 120s (with less CPU usage) and were omitted for space. Results grouped by makespan to show effects of more complex searches.

Use in StarCraft Playing Agents

Our planner (with macro actions) was incorporated into our StarCraft playing agent (*name removed*, written in C++ with BWAPI) which was previously a participant the 2010 AIIDE StarCraft AI Competition. When given expert knowledge goals, the agent was capable of planning to the goal in real time, executing the build order, and subsequently defeating some amateur level players, as well as the built-in StarCraft computer AI. The specific results are omitted since for this paper we are not concerned with the strength of the overall agent, but with showing that our build order planning system works in a real world competitive setting, something no existing method has accomplished.

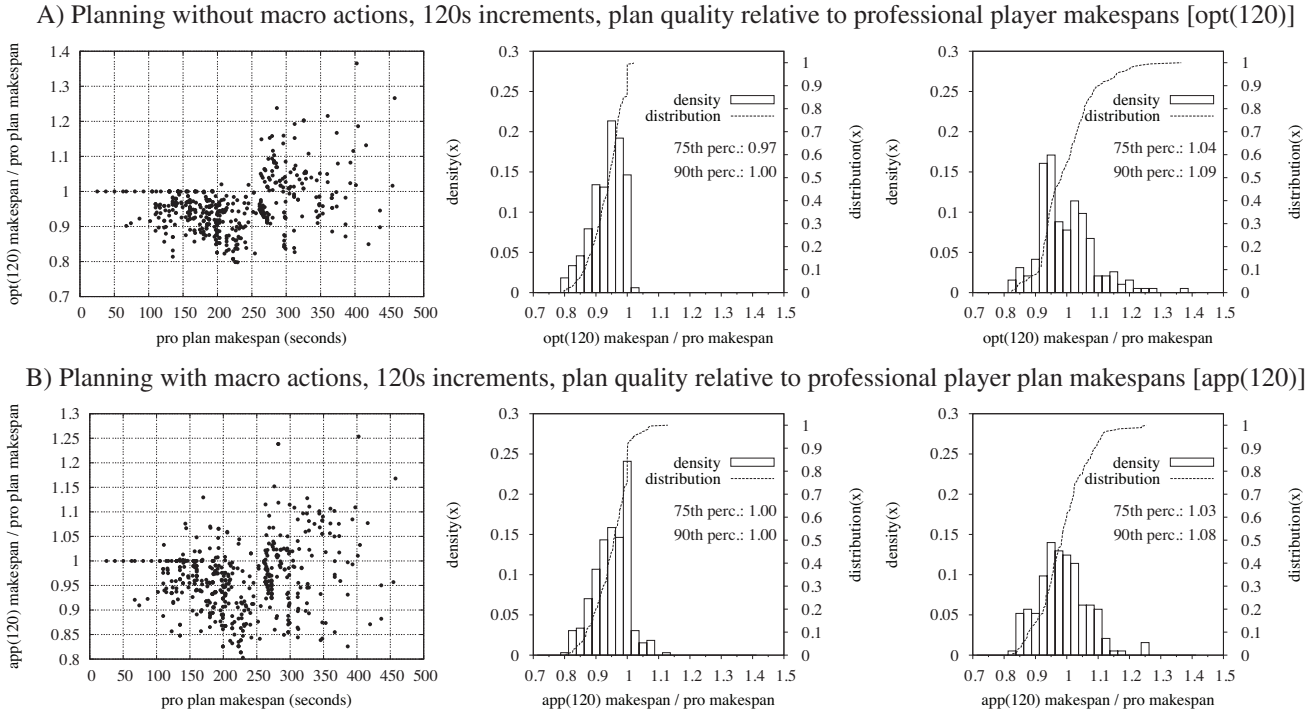


Figure 3: Makespan statistics for search without (A) and with (B) macro actions. Goals extracted by looking ahead 120s relative to professional player plan makespans. Shown are scatter plots of the makespan ratios (left), ratio densities, cumulative distributions, and percentiles for early game scenarios (pro makespan 0..249s, center) and early-mid game scenarios (250..500s). E.g. the top-middle graph indicates that 90% of the time, our planner produces makespans that match those of professionals.

Conclusion and Future Work

In this paper we have presented heuristics and abstractions that reduce the search effort for solving build order problems in StarCraft significantly while producing near optimal plans in real-time. We have shown macro actions, breadth limiting techniques, income abstractions, and multiple lower bound heuristics which reduce search spaces exponentially. A fast forwarding approach was introduced which replaced the null action, cut down on simulation time, and eliminated the need to solve the subset action selection problem.

We have shown that with all of these techniques, our planner is capable of producing plans in real-time which are comparable to professional StarCraft players, many of which have played the game for more than 10 years. We have also incorporated our solution into an actual game-playing agent which is capable of defeating non-trivial opponents, eliminating the need for the tedious hard-coding of build orders often present in similar agents.

In the future we plan to improve our techniques for dealing with more complex search goals by learning macro actions, and adding analysis of income data to further restrict search. Our abstractions (such as income) can also be improved by adjusting various parameters to reflect game context or specific environments. Our ultimate goal for the future of our planning system is the incorporation of strategic building placement and adversarial search. This would allow for goals such as defending a base or defeating enemy forces, eliminating the need for expert knowledge goals to be given to our planner, greatly improving the strength and adaptability of a StarCraft playing agent.

References

Branquinho, A., and Lopes, C. 2010. Planning for resource production in real-time strategy games based on partial order planning, search and learning. In *Systems Man and Cybernetics (SMC), 2010 IEEE International Conference on*, 4205–4211. IEEE.

Buro, M., and Kovarsky, A. 2007. Concurrent action selection with shared fluents. In *AAAI Vancouver, Canada*.

BWAPI. 2011. BWAPI: An API for interacting with StarCraft: Broodwar. http://code.google.com/p/bwapi/.

Chan, H.; Fern, A.; Ray, S.; Wilson, N.; and Ventura, C. 2007a. Extending online planning for resource production in real-time strategy games with search.

Chan, H.; Fern, A.; Ray, S.; Wilson, N.; and Ventura, C. 2007b. Online planning for resource production in real-time strategy games. In *Proceedings of the International Conference on Automated Planning and Scheduling, Providence, Rhode Island*.

Iba, G. 1989. A heuristic approach to the discovery of macro-operators. *Machine Learning* 3(4):285–317.

Kovarsky, A., and Buro, M. 2006. A first look at buildorder optimization in real-time strategy games. In *Proceedings of the GameOn Conference*, 18–22. Citeseer.

ORTS. 2010. ORTS - A Free Software RTS Game Engine. http://skatgame.net/mburo/orts/.

Stolle, M., and Precup, D. 2002. Learning options in reinforcement learning. *Abstraction, Reformulation, and Approximation* 212–223.