

Stochastic Plan Optimization in Real-Time Strategy Games

Andrew Trusty, Santiago Ontañón, Ashwin Ram

Cognitive Computing Lab (CCL)

College of Computing, Georgia Institute of Technology

Atlanta, Georgia, USA

atrusty@gatech.edu, {santi, ashwin}@cc.gatech.edu

Abstract

We present a domain independent off-line adaptation technique called Stochastic Plan Optimization for finding and improving plans in real-time strategy games. Our method is based on ideas from genetic algorithms but we utilize a different representation for our plans and an alternate initialization procedure for our search process. The key to our technique is the use of expert plans to initialize our search in the most relevant parts of plan space. Our experiments validate this approach using our existing case based reasoning system Darmok in the real-time strategy game Wargus, a clone of Warcraft II.

Introduction

Case based reasoning (CBR) systems solve new problems by reusing or adapting solutions that were used to solve past problems. Because of this, CBR systems are dependent on the quality of their cases in terms of their applicability to novel situations and their performance in different situations. In applying a CBR approach to the domain of real-time strategy (RTS) games, it is often true that the cases available are applicable in a very restricted set of situations and their performance is not always optimal due to the huge plan space and the non-deterministic nature of real-time strategy games. The problem of generating new cases or improving existing ones is quite apparent. Our team has previously performed research into adapting plans (Sugandh, Ontañón, & Ram 2008) to expand their applicability but in this paper we will present a more general domain independent off-line stochastic plan optimization technique which aims to adapt or completely replace plans in order to find or develop the best plan for any given situation.

Some of the typical approaches to adapting cases involve domain specific adaptation rules in CHEF (Hammond 1990), domain independent search methods used by PRIAR (Kambhampati & Hendler 1992), and case merging done in MPA (Ram & Francis 1996). In contrast to CHEF, our approach provides a domain independent but expert guided approach which can adapt to a wider range of situations than MPA while generally alleviating some of the time cost of PRIAR by using a heuristic function that guides the search.

Copyright © 2008, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.



Figure 1: A screenshot of the Wargus game.

The application of stochastic search based optimization to case based reasoning agents in real-time strategy games has the capability to discover useful new cases and expand the utility of existing ones. In our approach, we use this capability to perform plan optimization by stochastically searching the plan space. We start with a plan to optimize and use this plan as an initial direction and branching point in our optimization process which iteratively explores and exploits the plan space using in-game evaluations of generated plans as a guide. We empirically validate our process by integrating it into our existing case based reasoning system, Darmok (Ontañón *et al.* 2007), and testing it in novel situations where the system lacks cases to perform intelligently.

The outline of the rest of the paper is as follows. In the first section we discuss the importance and challenge of the domain of RTS games. The second section discusses related work. Next we provide a review of the existing Darmok system architecture and how our new plan optimization integrates with it. The following section covers the Stochastic Plan Optimization algorithm (SPO). The sixth section gives an example of a situation where the algorithm can be successfully deployed. The seventh section presents an empirical evaluation. Finally, we present our conclusions.

Real-Time Strategy Games

Wargus (Figure 1) is a clone of the classic real-time strategy game Warcraft II where each player's goal is to survive and destroy the other players. Each player has a number of troops, buildings, and workers who gather resources (gold, wood and oil) in order to produce more units. Buildings are required to produce more advanced troops, and troops are required to attack the enemy. The calculations inherent in the combat system make the game non-deterministic. For example, the map shown in Figure 1 leads to complex strategic reasoning, such as building long range units (such as catapults or ballistas) to attack the other player before the wall of trees has been chopped down, or chopping through early in the game to try to catch the enemy by surprise.

Real-time strategy games have been recognized as domain rich in interesting problems for artificial intelligence researchers (Buro 2003; Aha, Molineaux, & Ponsen 2005). They offer a variety of challenges given their huge state space, their non-deterministic nature, the partially observable environments, and the presence of multiple adversaries with unknown decision models. The huge commercial stake the game industry has in improving the intelligence embedded in their games and the growing audience of strategy gamers provides a ripe domain for artificial intelligence research to have a dramatic effect.

Related Work

One of the first case-based planning systems was CHEF (Hammond 1990). CHEF was able to build new recipes with multiple goals based on the user's request for dishes with particular ingredients and tastes. CHEF contained a memory of past failures to warn about problems and also a memory of succeeded plans from which to retrieve plans. One of the novel capabilities of CHEF, with respect to classical planning systems, was its ability to learn. The places in which CHEF experienced planning failures were the places the system needed to learn. CHEF performed plan adaptation by a set of domain-specific rules called TOPs.

Domain-independent nonlinear planning has been shown to be intractable (NP-hard). PRIAR (Kambhampati & Hendler 1992) was designed to address that issue. PRIAR works by annotating generated plans with a *validation structure* that contains an explanation of the internal causal dependencies so that previous plans can be reused by adapting them in the future. Related to PRIAR, the SPA system was presented by Hanks and Weld (Hanks & Weld 1995). The key highlight of SPA is that it is complete and systematic (while PRIAR is not systematic, and CHEF is neither complete or systematic), but uses a simpler plan representation than PRIAR. Extending SPA, Ram and Francis (Ram & Francis 1996) presented MPA (Multi-Plan Adaptor), which extended SPA with the ability to merge plans. These approaches require that the domain is specified in a particular form whereas our approach does not. For an extensive overview of case-based plan adaptation techniques see (Muñoz-Avila & Cox 2007).

There are several reasons for which traditional search-based planning approaches cannot be directly applied to do-

main such as Wargus. The first one is the size of the decision space. If we follow the analysis performed in (Aha, Molineaux, & Ponsen 2005), the approximate number of different commands that can be issued in the situation depicted in Figure 1 is about 280,000. Thus, classical adversarial search using a mini-max kind of algorithm is not feasible. We will discuss later how our approach addresses this challenge by using expert cases to guide the search.

Ponsen and Spronck have done the most similar work in their evolutionary learning approach applied to their Dynamic Scripting technique (Spronck *et al.* 2006). Like our work, theirs is an off-line evolutionary algorithm which automatically generates tactics for Wargus. Their work showed the viability of using evolutionary learning to aid adaptive AI systems in real-time strategy games. Their approach differs from ours in two important ways, they used a chromosome based tactic representation and randomly initialized their evolutionary search process.

Our approach also shares many of the characteristics of a standard genetic algorithm (Koza 1998). We differ most importantly in how we initialize our population using expert cases, in the way we decay probabilities involved in the application of our operators, and the fact that we do not use a gene-like representation. In this sense, our approach could be viewed as an amalgam between a genetic algorithm and simulated annealing with an expert guided selection of the initial population.

Darmok

In this section we will briefly describe Darmok, in which we have implemented our plan optimization techniques and how our optimization system fits into the Darmok architecture (see (Ontañón *et al.* 2007) for more details about Darmok).

Darmok learns behaviors from expert demonstrations and uses case-based planning techniques to reuse the behaviors for new situations. Basically, Darmok's execution can be divided in two main stages:

- *Behavior acquisition:* During this first stage, an expert plays a game of Wargus and the trace of that game is stored. Then, the expert annotates the trace explaining the goals he was pursuing with the actions he took while playing. Using those annotations, a set of behaviors are extracted from the trace and stored as a set of cases. Each case is a triple: situation/goal/behavior, representing that the expert used a particular behavior to achieve a certain goal in a particular situation.
- *Execution:* The execution engine consists of several modules, which together maintain a current plan to win the game. The *Plan Execution* module is in charge of executing the current plan and update its state (marking which actions succeeded or failed). The *Plan Expansion* module is in charge of identifying open goals in the current plan and expanding them. In order to do that, it relies on the *Behavior Retrieval* module, which given an open goal and the current game state will retrieve the most appropriate behavior to fulfill that open goal. Finally, we have the *Plan Adaptation* module which is in charge of adapting the retrieved plans according to the current game state.

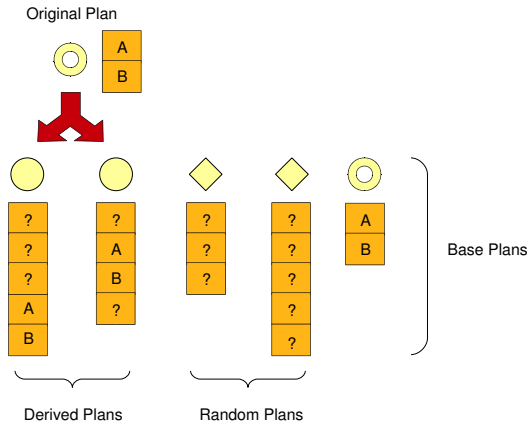


Figure 2: Visualizing the generation of the initial base plans.

Our plan optimization system intercepts the retrieval of the specific case we are trying to optimize in order to generate and return plan variants for the system to run. We evaluate our variants after the *Plan Execution* module runs the plans, at which point it reports back whether the plan succeeded or failed. In the remainder of this paper we will focus on the plan optimization technique we have designed to perform offline plan optimization.

Plan Optimization

In this section we will discuss the design of our plan optimization engine. Our design is based on a couple of assumptions which define our search strategy. First, searching the entire plan space is an intractable problem and so we will limit the scope of our search. Second, the expert trace may not be the best but is all the information we have in order to know where to start the search, so we have built in the ability to decide how much to trust the expert.

Our stochastic search strategy process realizes these assumptions in two stages which we continually iterate through, *plan generation* and *plan evaluation*. Figure 4 presents the algorithm. We will explain each part of it in detail.

Plan Generation

There are two different phases of plan generation. For the first iteration only, we generate a pre-defined number of base plans to seed our system. These base plans are either derived from the original expert generated plan or completely random base plans, see Figure 2 for a visualization of this process. The more derived plans used, the more we trust the expert and our search will be centered closer in plan space to the original plan. Conversely, more random plans indicate less trust in the expert and will cause a more diverse plan space to be explored. The balance between the number of random and derived plans provides a nice control for users to decide whether they want to concentrate on improving or replacing the existing plan. Each of these base plans are sequential or parallel plans which are randomly padded with a random number of randomly generated plans. The range

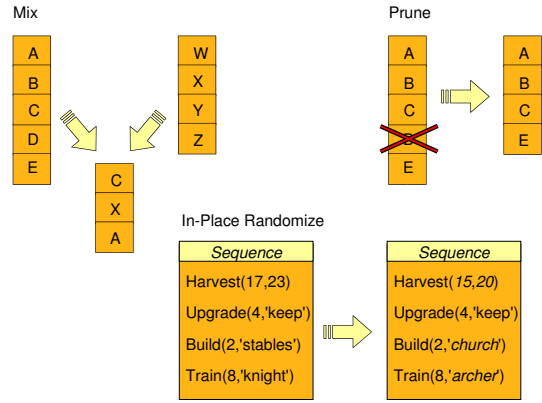


Figure 3: Visualizations for the Mix, Prune, and In-Place Randomize plan generation operators. The randomize operator is not shown because its output has no parent derived relation.

of the number of padded random plans is an input which provides an approximate upper bound on the size of plans considered in our search. We will see later how the plans are able to grow further but will tend to shrink in size due to the nature of our plan generation operators. The base plans in our initial phase of plan generation let us start our search in fertile ground where we have a lot of wiggle room to exploit the expert plan or explore the plan space further.

The second phase of plan generation is run every iteration. It tries to optimize the plans that survived the previous iteration by using a number of different plan generation operators. There are four plan generation operators in our system. Together, these operators attempt to provide a balance between exploration and exploitation of the previous iterations plans. The explore/exploit balance can be customized by setting probabilities for each operator which define roughly how often they will be applied. Currently, we implement the following four operators: *randomize*, *mix*, *prune*, *in-place randomize*. See Figure 3 for visualizations of the *mix*, *prune* and *in-place randomize* operators.

- The *randomize* operator creates a completely new and random plan. Randomize is the most exploration centered operator given that it effectively jumpstarts a new search direction unrelated to the existing directions.
- The *mix* operator provides slightly less exploration and at the same time tries to exploit two different randomly chosen plans which are mixed together to generate a new plan. Mixing is the only operator which can create larger plans than the initial base plans by combining steps from both plans.
- The *prune* operator does the least exploration by removing a single step from a plan in the hopes of down-sizing the plan to its critical steps.
- The *in-place randomize* operator randomly changes the subplans of a plan without changing the ordering within the plan.

Plan Evaluation

Once the plans have been run in-game we use an evaluation heuristic to generate scores (or fitness function in genetic algorithms terminology). Only plans with a high score will be used in the next iteration. The evaluation heuristic is based mainly on the resource value of all of a player's units, technologies researched, and a concept of territorial control based on troop and building locations. A score for each player is calculated as follows:

$$100 * tr + \sum resources + \sum_{military\ units} h(unit) * rc(unit) * 1.5 + \sum_{civilian\ units} h(unit) * rc(unit) + \sum_{technology} rc(r)$$

Player controlled territory, the area of the map within a distance of 3 or less to the players units, is represented by tr . The rc function returns the sum of the amount of each resource (gold, wood, oil) required to construct a unit. The h function returns a unit's current health divided by their max health. The value of military troops is scaled up to emphasize their strategic value and the value of all units is proportional to their health to account for uncompleted or damaged buildings and injured troops. The final score for a plan is the sum of the enemy's scores subtracted from the players' score. This scoring system attempts to favor plans which improve the players success at the expense of the enemies.

Every iteration we keep only the best N plans which are used to generate the next crop of plans. After all the iterations are completed, we retain all the plans which are some pre-defined percent better than the original plan.

Example

In this section we will give an example of a situation where SPO could help. Imagine a scenario in which Darmok has a fighter and a ballista and the enemy has a fighter and a tower. The ballista works well against the tower because it can destroy the tower without getting in range of the tower's attack but the enemy fighter stationed near the tower can destroy the ballista easily when it attacks either the tower or the fighter. Also, the fighter cannot destroy the tower and only has a 50% chance of defeating the enemy fighter. The optimal plan would have our agent use the ballista and the fighter to attack the enemy fighter and then have the ballista destroy the tower on its own. This plan is not in our case base though. Our system must therefore adapt and optimize the attack tower plan to include two attack plans for our fighter and ballista to attack the enemy fighter before the ballista attack tower plan. Figure 5 shows a small part of the trace of our search process in this example.

Experimental Results

We ran experiments on four maps. Maps 1 and 1b model the situation described in the above example with one having greater complexity with the addition of more units and buildings. Maps 2 and 2b, one simple and one complex with

Function $optimize(o, s, d, nR, nE, nS, nI, sP, \vec{P}, \vec{D})$:

```

plans = empty
plans.addAll(generate d base plans derived from o)
plans.addAll(generate nR random base plans)
add [0, nE] subplans to every plan in plans
For every plan in plans: plan.subplans.shuffle()
untriedPlans = plans.copy()
Repeat nI times:
  For every plan in plans:
    For every operator:
      untriedPlans.add(apply(operator, plan, P))
  For every plan in untriedPlans:
    score = evaluatePlan(plan)
    plans.add(plan, score)
  keep only the nS highest scoring plans in plans
  untriedPlans = empty
  P *= D
retain plans with scores sP % better than s to case base

```

Figure 4: The SPO algorithm, where o is the original plan, s is the original plan's score, d is the number of derived plans, nR is the number of random plans, nE is the max number of plans to expand, nS is the number of plans to save each iteration, nI is the number of iterations, sP is the cutoff percent for saving the final plans, \vec{P} is the vector of probabilities for applying the operators, and \vec{D} is the vector of decay rates for the probabilities in \vec{P} .

more units and buildings, involve peasants foolhardily cutting down the forest protecting them from the enemy. The goal plan for maps 2 and 2b is any plan which prevents the peasants from cutting down the forest.

On each map we used three different algorithms, the SPO algorithm and two other algorithms used for comparison. The second algorithm is a variant of SPO we will call Random Base Plans. It works in exactly the same way as SPO but is only initialized with random base plans and not with derived base plans. The third algorithm we call Random Only because it only generates random plans. To most effectively compare the algorithms, we ran SPO for 10 iterations and then proceeded to run the other two for as long as it took them, with a max run-time limit, to achieve the same or better performance than SPO. The parameters for SPO were set as follows: $d = 3$, $nR = 3$, $nE = 2$, $nS = 6$, $nI = 10$, $sP = .1$, $\vec{P} = \{1.0, 1.0, 1.0, 1.0, 1.0\}$, $\vec{D} = \{.96, .98, .99, .98, .99\}$.

Figure 6 shows the results of running the algorithms on each map plotted in terms of their run-time and final score. To perform the experiments in a sensible time the run-time for each algorithm on maps 1 and 1b was limited to 2 hours and 4 hours on maps 2 and 2b. Maps 2 and 2b required more time for each plan to better evaluate the consequences of the plans. These experiments were run on a Pentium D 3.2GHz with 3GB of RAM.

On map 1 we can see that SPO provides a good balance of time for performance. Random Base Plans performs sim-

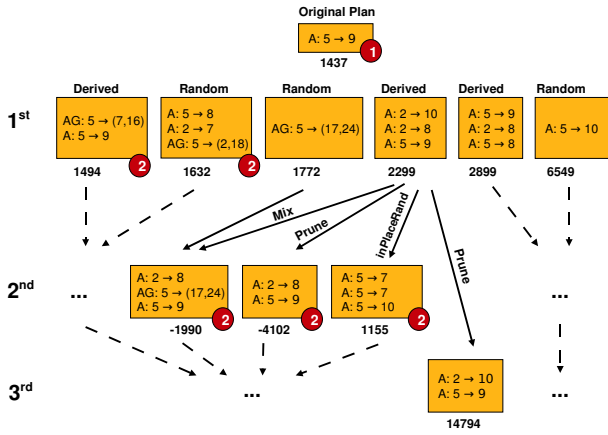


Figure 5: A partial trace of the search process in our example. The three iterations are labeled on the left. Each box represents a plan and the arrows represent child-parent relationships. The arrows are labeled with the operation used on the parent to generate the child. The numbers below the boxes are the plan scores and the numbers in red circles represent the iteration in which the plan was pruned from the search process.

ilarly but ends up trading a good chunk of time for performance on several runs and hits the max run-time a couple times. As could be expected random is all over the graph with undependable performance scores.

On map 1b, the more complex version of map 1, SPO takes a clear lead over the other two algorithms. Random Base Plans does reasonably well overall but is not able to match the scores of SPO and runs out of time a couple times. Random shows that it handles the complexity the worst and more often than not runs out of time. A further breakdown of the algorithms can be seen in Figure 7 which shows a breakdown of the best score overall from each individual run of each algorithm for every iteration. Here we can clearly see how random incrementally jumps in score at random intervals. It also shows that Random Base Plans is able to improve using the same adaptation techniques as SPO but lacks the derived base plans which enable SPO to attain the higher scores.

Maps 2 and 2b are interesting in that the plan provided to SPO is not an expert plan. In fact the plan is one of the worst plans possible in the situation. In this light it is easier to see why Random Base Plans which disregards the original plan performs better on 2 and 2b. SPO still does better on average than random but seems to take a while to forget the so-called expert plan.

Conclusion

Our experiments show that SPO is able to successfully develop new plans and adapt existing ones. It stands out as the most dependable algorithm among the three tested and provides the most balanced time and score tradeoff in situations which have a valid expert plan to build on. The performance of the algorithm depends upon the quality of the provided

expert plan. A good expert plan can speed up the search and guide it to the most fertile plan space whereas a bad plan can slow it down a good bit. Even in situations with a bad plan, the parameters to SPO can be specified to so as to not rely on the bad plan and instead perform more like the Random Base Plans algorithm performs.

There are a number of future research directions that are apparent in this work. Effectively, SPO concentrates on winning battles and not wars. SPO was designed to optimize individual plans used by the Darmok system, and thus, further experiments will evaluate the performance of the whole Darmok system when SPO is used as a subroutine to optimize plans when Darmok detects that a plan is suboptimal. The addition of domain knowledge to the calculations of the algorithm may be able to speed up the search process at the price of losing its applicability to any domain. It would also be interesting to experiment with the effects of multiple original expert plans instead of only one. Additional expert plans could offer better guidance for the initial plans which seed the search process.

References

- Aha, D.; Molineaux, M.; and Ponsen, M. 2005. Learning to win: Case-based plan selection in a real-time strategy game. In *ICCBR'2005*, number 3620 in LNCS, 5–20. Springer-Verlag.
- Buro, M. 2003. Real-time strategy games: A new AI research challenge. In *IJCAI'2003*, 1534–1535. Morgan Kaufmann.
- Hammond, K. F. 1990. Case based planning: A framework for planning from experience. *Cognitive Science* 14(3):385–443.
- Hanks, S., and Weld, D. S. 1995. A domain-independent algorithm for plan adaptation. *Journal of Artificial Intelligence Research* 2:319–360.
- Kambhampati, S., and Hendler, J. A. 1992. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence* 55(2):193–258.
- Koza, J. R. 1998. Genetic programming. In Williams, J. G., and Kent, A., eds., *Encyclopedia of Computer Science and Technology*, volume 39, 29–43. Marcel-Dekker.
- Muñoz-Avila, H., and Cox, M. 2007. Case-based plan adaptation: An analysis and review. *IEEE Intelligent Systems*.
- Ontañón, S.; Mishra, K.; Sugandh, N.; and Ram, A. 2007. Case-based planning and execution for real-time strategy games. In *Proceedings of ICCBR-2007*, 164–178.
- Ram, A., and Francis, A. 1996. Multi-plan retrieval and adaptation in an experience-based agent. In Leake, D. B., ed., *Case-Based Reasoning: Experiences, Lessons, and Future Directions*. AAAI Press.
- Spronck, P.; Ponsen, M.; Sprinkhuizen-Kuyper, I.; and Postma, E. 2006. Adaptive game ai with dynamic scripting. *Mach. Learn.* 63(3):217–248.
- Sugandh, N.; Ontañón, S.; and Ram, A. 2008. On-line case-based plan adaptation for real-time strategy games. In *AAAI'2008*, to appear.

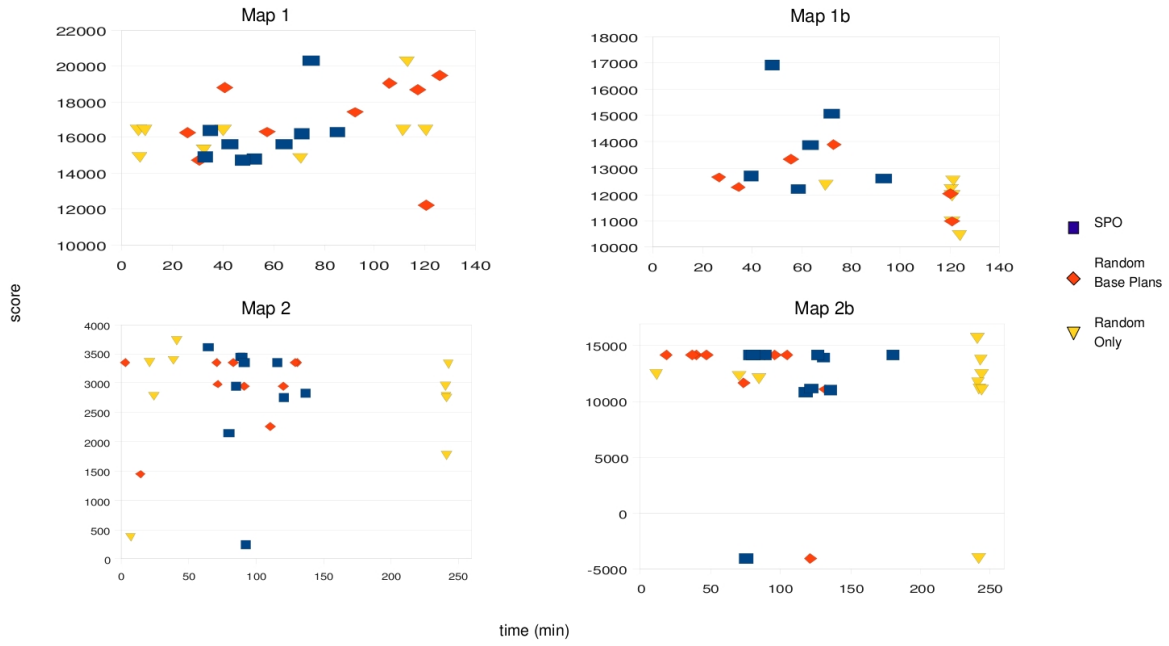


Figure 6: Final time and score results achieved by each run of each algorithm on the four maps.

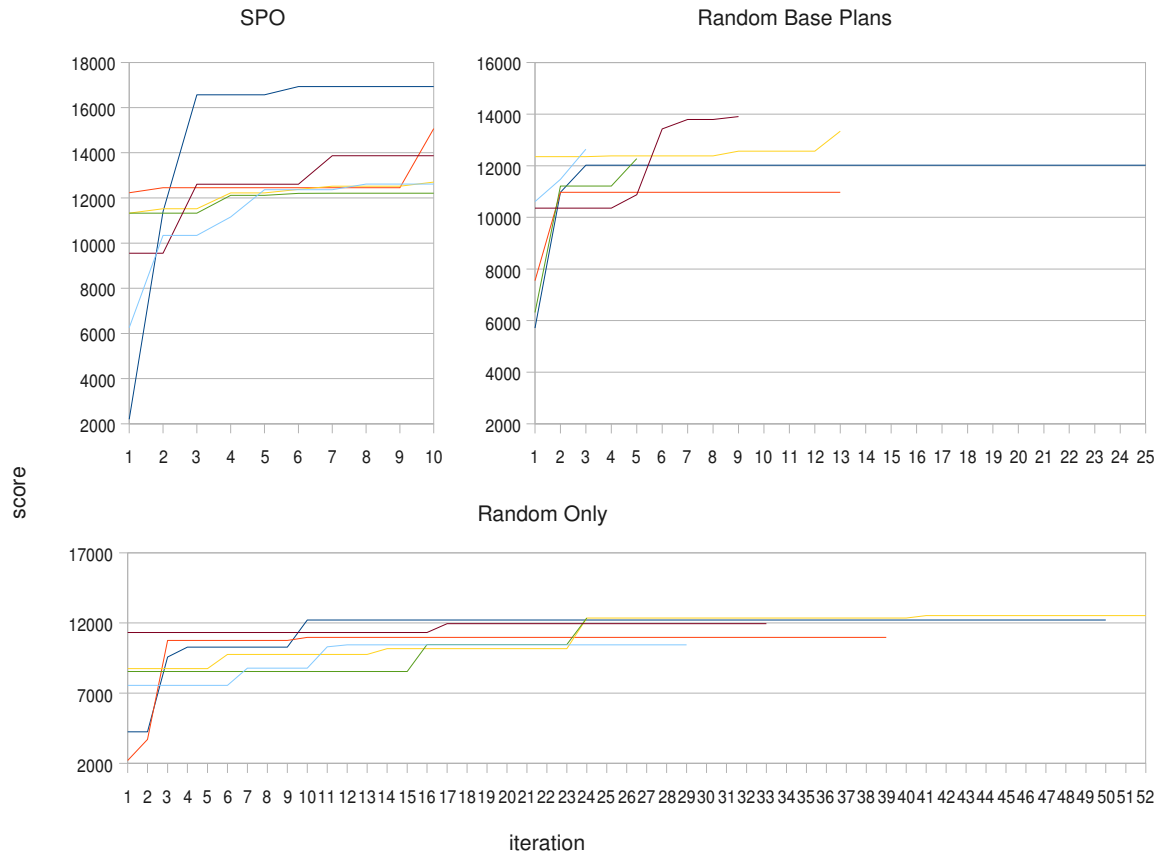


Figure 7: Best score overall from each individual run of each algorithm for every iteration on map 1b.