Graph search has been employed by many AI techniques and applications. A natural way to improve the efficiency of search is to utilize advanced, more powerful computing platforms. However, expensive computing infrastructures, such as supercomputers and large-scale clusters, are traditionally available to only a limited number of projects and researchers. As a result, most AI applications, with access to only commodity computers and clusters, cannot benefit from the efficiency improvements of high-performance parallel search algorithms. Cloud computing provides an attractive, highly accessible alternative to other traditional high-performance computing platforms. In this paper, we first show that the run-time of our stochastic search algorithm in planning is a heavy-tailed distribution, which has a remarkable variability. Second, we propose an algorithm framework that takes advantage of cloud computing.

... Read complete abstract on page 2.
Cloud Computing for Scalable Planning by Stochastic Search

Complete Abstract:

Graph search has been employed by many AI techniques and applications. A natural way to improve the efficiency of search is to utilize advanced, more powerful computing platforms. However, expensive computing infrastructures, such as supercomputers and large-scale clusters, are traditionally available to only a limited number of projects and researchers. As a result, most AI applications, with access to only commodity computers and clusters, cannot benefit from the efficiency improvements of high-performance parallel search algorithms. Cloud computing provides an attractive, highly accessible alternative to other traditional high-performance computing platforms. In this paper, we first show that the run-time of our stochastic search algorithm in planning is a heavy-tailed distribution, which has a remarkable variability. Second, we propose an algorithm framework that takes advantage of cloud computing.
Cloud Computing for Scalable Planning by Stochastic Search

Authors: Qiang Lu and You Xu and Ruoyun Huang and Yixin Chen

Abstract: Graph search has been employed by many AI techniques and applications. A natural way to improve the efficiency of search is to utilize advanced, more powerful computing platforms. However, expensive computing infrastructures, such as supercomputers and large-scale clusters, are traditionally available to only a limited number of projects and researchers. As a result, most AI applications, with access to only commodity computers and clusters, cannot benefit from the efficiency improvements of high-performance parallel search algorithms. Cloud computing provides an attractive, highly accessible alternative to other traditional high-performance computing platforms. In this paper, we first show that the run-time of our stochastic search algorithm in planning is a heavy-tailed distribution, which
ied and applied to several areas of automated planning, such as sampling possible trajectories in probabilistic planning (Bryce, Kumhampati, and Smith 2006) and robot motion planning (LaValle 2006). (Fern, Yoon, and Givan 2004) uses random walk exploration to lean domain-specific control knowledge.

This paper generally has two contributions. First, we show that the run-time distribution Monte Carlo Random Walk (MRW) algorithm in planning is a heavy-tailed distribution, which has a remarkable variability. Second, we propose a parallel MRW algorithm which takes advantage of short runs in thus heavy-tailed distribution. Our parallel MRW algorithm is a parallel stochastic search which use low frequency communication, even no communication between computing nodes which is perfectly suitable for cloud computing architecture.

The remainder of this paper is organized as follows: Section briefly reviews the cloud computing, SAS+ formalism of classic planning and Monte Carlo Random Walk method. Section explains details of MRW algorithm and its heavy-tailed run-time distribution. Section introduces basic parallel MRW algorithm and a communication technique to improve the efficiency and probability of solving problems. Section discusses the performance on standard planning benchmarks from the IPC-4 competition. Section contains concluding remarks and some potential directions for future work.

Background

Cloud computing

SAS+ Formalism

In this paper, we work on the SAS+ formalism (Jons- son and Bäckström 1998) of classical planning. In the following, we review this formalism and introduce our notations.

Definition 1. A SAS+ planning task $Π$ is defined as a tuple \{X, O, S, s_I, s_G\}.

- $X = \{x_1, \cdots, x_N\}$ is a set of multi-valued state variables, each with an associated finite domain $\text{Dom}(x_i)$.
- $O$ is a set of actions and each action $o \in O$ is a tuple (pre($o$), eff($o$)), where both pre($o$) and eff($o$) define some partial assignments of variables in the form $x_i = v_i, v_i \in \text{Dom}(x_i)$. $s_G$ is a partial assignment that defines the goal.
- $S$ is the set of states. A state $s \in S$ is a full assignment to all the state variables. $s_I \in S$ is the initial state. A state $s$ is a goal state if $s_G \subseteq s$.

For a given state $s$ and an action $o$, when all variable assignments in pre($o$) are met in state $s$, action $o$ is applicable at state $s$. After applying $o$ to $s$, the state variable assignment will be changed to a new state $s'$ according to eff($o$): the state variables that appear in eff($o$) will be changed to the assignments in eff($o$) while other state variables remain the same. We denote the resulting state of applying an applicable action $o$ to $s$ as $s' = \text{apply}(s, o)$. apply($s, o$) is undefined if $o$ is not applicable at $s$. The planning task is to find a plan, a sequence of actions that transits the initial state $s_I$ to a goal state that includes $s_G$.

An important structure for a given SAS+ task is the domain transition graph defined as follows.

Definition 2. For a SAS+ planning task, each state variable $x_i, i = 1, \cdots , N$ corresponds to a domain transition graph (DTG) $G_i$, a directed graph with a vertex set $V(G_i) = \text{Dom}(x_i) \cup v_0$, where $v_0$ is a special vertex, and an edge set $E(G_i)$ determined by the following.

- If there is an action $o$ such that $(x_i = v_i) \in \text{pre}(o)$ and $(x_i = v_i') \in \text{eff}(o)$, then $(v_i, v_i') \in E(G_i)$ and we say that $o$ is associated with the edge $e_i = (v_i, v_i')$ (denoted as $o \vdash e_i$). It is conventional to call the edges in DTGs as transitions.
- If there is an action $o$ such that $(x_i = v_i') \in \text{eff}(o)$ and no assignment to $x_i$ is in $\text{pre}(o)$, then $(v_0, v_i') \in E(G_i)$ and we say that $o$ is associated with the transition $e_i = (v_0, v_i')$ (denoted as $o \vdash e_i$).

Intuitively, a SAS+ task can be decomposed into multiple objects, each corresponding to one DTG, which models the transitions of the possible values of that object.

Monte-Carlo Random Walk

In Monte-Carlo Random Walk planning (Nakhost and Miller 2009), fast Monte-Carlo random walks are used for exploring the neighborhood of a search state. A relatively large set of states $S$ in the neighborhood of the current state $s_0$ is sampled before greedily selecting a most promising next state $s \in S$. For example, a new random walk starts from $s_0$, builds a sequence of actions $o_0 \rightarrow o_1 \rightarrow \cdots \rightarrow o_k$ and changes $s_0$ to $s$. At the end of the random walk, $s$ is evaluated by a heuristic function $h$, for instance by the FF heuristic, and added to $S$. When a stopping criterion is satisfied, the algorithm chooses a state in $S$ with the minimum $h$-value to replace $s_0$.

The MRW method uniformly deals with both problems of local search methods: it quickly escapes from local minima and can recover from areas where the evaluation is poor. The MRW method does not rely on any assumptions about the local properties of the search space or heuristic function.

Monte-Carlo Random Walk Search

Algorithm 1 shows the framework of Monte-Carlo Random Walk method. Given a SAS+ planning problem $Π$, MRW search builds a chain of states $s_I \rightarrow s_1 \rightarrow \cdots \rightarrow s_n$ such that $s_I$ is the initial state, $s_n$ is a goal state, and each transition $s_i \rightarrow s_{i+1}$ uses an action sequence found by RandomWalk exploring the neighborhood of $s_i$ (Line 9). MRW search fails to find a solution when the minimum obtained $h$-value does not improve within MAX_STEPS times, or $s_i$ is a dead-end.
### Table 1: Expected speedup of parallel MRW Algorithm.

<table>
<thead>
<tr>
<th>problem</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>air-17</td>
<td>4.76</td>
<td>7.64</td>
<td>11.40</td>
<td>15.73</td>
<td>19.83</td>
<td>22.92</td>
</tr>
<tr>
<td>tank-31</td>
<td>4.94</td>
<td>7.94</td>
<td>11.86</td>
<td>16.30</td>
<td>20.56</td>
<td>24.08</td>
</tr>
<tr>
<td>notan-45</td>
<td>4.76</td>
<td>7.64</td>
<td>11.40</td>
<td>15.73</td>
<td>19.83</td>
<td>22.92</td>
</tr>
<tr>
<td>sate-24</td>
<td>1.20</td>
<td>1.50</td>
<td>1.76</td>
<td>1.94</td>
<td>2.00</td>
<td>2.00</td>
</tr>
</tbody>
</table>

Parallel MRW Planning

Based on the study of heavy-tailed distributions, we present a parallel MRW procedure expecting to take advantage of short runs and significantly reduce solving time. Algorithm 2 shows the framework of parallel MRW procedure (PMRW). It simply use $N$ processes to run MRW procedure independently (Line 2). The procedure will abort all other processes when a process find a solution (Line 4). Obviously, the solve time of the PMRW is the minimal solve time of $N$ independent runs of MRW.

Suppose $X$ and $X'$ are the runtime variable of MRW and PMRW. We have

$$P(X' < x) = 1 - [1 - P(X < x)]^N \quad (1)$$

Let $EX$ be the mean of $X$. Suppose $P(X < EX) = 0.2$ and $N = 8$, according to formula (1), $P(X' < EX) = 0.83$. Thus, even though the probability of short runs are low in MRW, the probability of hitting the same short runs are high enough to be accepted in PMRW. We can compute the expected mean of $X'$ according to the runtime distribution of $X$. Table 1 gives the expected speedup($EX/EX'$) with different $N$ of four instances.
Algorithm 3: MRW-C(II, Q)

Input: SAS+ planning problem II, Queue Q
Output: a solution plan
1. $s \leftarrow s_I$;
2. $plan \leftarrow \emptyset$;
3. $h_{min} \leftarrow h(s_I)$;
4. $counter \leftarrow 0$;
5. while $s$ does not satisfy $s_Q$ do
   if $counter > MAX\_STEPS$ or DeadEnd($s$) then
      plan, $s \leftarrow \text{ReadMessage}(Q)$;
      $m\_counter \leftarrow 0$;
      $counter \leftarrow 0$;
   else
      $plan_{temp}, s_{temp} \leftarrow \text{RandomWalk}(s, II)$;
      if $h(s_{temp}) < h_{min}$ then
         if $m\_counter < MAX\_M$ then
            $WriteMessage(Q, plan_{temp}, s_{temp})$;
            $m\_counter \leftarrow m\_counter + 1$;
         else
            $plan \leftarrow plan_{temp}$;
            $s \leftarrow s_{temp}$;
            $h_{min} \leftarrow h(s)$;
            $counter \leftarrow 0$;
         else
            $plan \leftarrow plan_{temp}$;
            $s \leftarrow s_{temp}$;
            $counter \leftarrow counter + 1$;
   end
end
return plan;

MRW procedure intergrading communication technique (MRW-C). Algorithm 3 has two changes compared to Q. First, when MRW stuck to a bad run, not getting any progress within MAX\_STEPS walks or $s_i$ is a dead-end state (Line 6), it will require a message from Q. If $Q$ has messages, RequireMessage() will return a tuple $(plan, s)$ which is a possible good search candidate. Otherwise, it will return $(\emptyset, s_I)$. Second, when MRW finds a better state, it will write a message to $Q$ (Line 13). The number of messages written by a good run is at most MAX\_M. This bound make sure that a good run will search forward after writing some good states.

There are two issues of communication technique in our experiments. One is when requiring message from $Q$, there are a lot of states having the same best heuristic value. Another is by replacing bad runs with good runs, some processes may search in the same local area of search space. It make some of them do repeat and no use work. We introduce a hamming distance $H$ to address the above problems (Hamming 1950). The hamming distance between two states $s_1$ and $s_2$ is defined as:

$$H(s_1, s_2) = \sum_{\forall x_i \in X} d(s_1(x_i), s_2(x_i)).$$

$d(s_1(x_i), s_2(x_i))$ is the distance of $s_1(x_i)$ and $s_2(x_i)$ in $x_i$’s domain transition graph $G_i$.

| Figure 1: The distribution of solving time in different instances. | (a) airport-17 | (b) tankage-31 | (c) no-tankage-45 | (d) satellite-24 |

Conclusions

References


Figure 2: Solution time on IPC4 domains.

Table 2: Comparison of 1-core, 8-core baseline and 8-core Monte Carlo Algorithms.


