Stochastyczne Algorytmy Obliczeniowe

Zastosowanie algorytmu genetycznego do rozwiązania problem replikacji danych w środowisku rozproszonym

Andrzej Kaczmarczyk Marcin Łoś

1 Wstęp

Celem projektu było wybranie jednego problemu obliczeniowego, zapoznanie się z istniejącymi jego rozwiązaniami, oraz próba ulepszenia któregoś z nich. Nasz wybór padł na problem replikacji danych w systemach rozproszonych, oraz rozwiązanie oparte na algorytmie genetycznym, opisane w [1].

2 Opis problemu

Dany jest system rozproszony, składający się z M hostów $\{H_i\}$, o pojemnościach s_i, połączonych siecią komunikacyjną tak, że koszt przesyłu jednostki danych pomiędzy H_i i H_j wynosi $C(i,j)$. Istnieje także N obiektów $\{O_i\}$, o rozmiarach o_i. Host H_i wykonuje na obiekcie O_k odpowiednio $r_k^{(i)}$ operacji odczytu, i $w_k^{(i)}$ operacji zapisu. Każdy obiekt O_i znajduje się pierwotnie na jednym hoście, SP_i.

Obiekty mogą zostać zreplikowane na inne hosty, tak, że suma rozmiarów obiektów zreplikowanych na hoście H_i nie przekracza jego pojemności s_i. Wszystkie hosty mają pełną wiedzę o replikach obiektów. Operacja odczytu obiektu O_k z hosta H_i przebiega w ten sposób, że obiekt jest wysyłany do hosta H_i z najbliższego mu hosta zawierającego replikę O_k. Operacja zapisu natomiast odbywa się w ten sposób, że host H_i wysyła nowy stan obiektu do hosta SP_k (pierwotnego miejsca zwierającego O_k), a ten rozsyła informację o zmianie do pozostałych hostów zawierających repliki O_k.

Koszt sumaryczny przy danym rozłożeniu replik to suma kosztów przesyłania obiektów spowodowanego operacjami odczytu i zapisu, przebiegającymi w opisany powyżej sposób. Koszt pojedynczego przesyłu to iloczyn ilości przesyłanych danych (rozmiaru obiektu) i kosztu jednostkowego przesyłu między hostami (danego przez $C(i,j)$). Problem polega na znalezieniu replikacji minimalizującej koszt sumaryczny.

Nieco bardziej szczegółowy opis, wraz z wzorami na całkowity koszt znaleźć można w [1].

3 Istniejące rozwiązania

4 Rozwiązanie bazowe

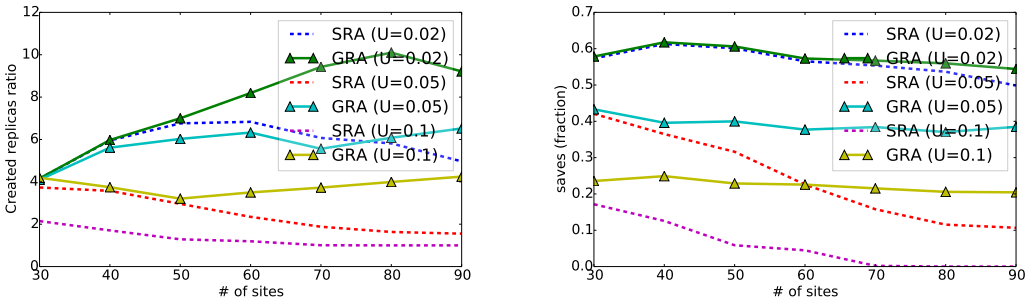
Jako punkt wyjściowy przyjęliśmy rozwiązanie zaproponowane w [1].

5 Narzędzia

Do realizacji implementacyjnych aspektów projektu wykorzystaliśmy język Python i bibliotekę PyEvolve [2], dostarczającą różnych komponentów (np. strategie selekcji) pozwalających budować algorytmy genetyczne. Do implementacji rozproszonej użyte zostało MPI, za pośrednictwem Pythonowych bindingów udostępnianych przez bibliotekę mpi4py [3].

6 Przebieg prac

Pierwszym celem po dokładnym zapoznaniem się z artykułem, na którym bazuje projekt, było odtworzenie zaprezentowanych w nim wyników. W tym celu zaimplementowany został dokładnie przedstawiony w nim algorytm (zarówno deterministyczny zachłanny – SRA – jak i główny – GRA). Początkowo planowaliśmy użyć nieco innego generatora danych do testów, takiego, który w naszym odczuciu mógłby lepiej odwzorowywać własności instancji problemu, które występować mogą w praktyce. Dobranie jednak parametrów tak, by zaobserwować wyniki podobne do tych przedstawionych w artykule bazowym okazało się trudne, w związku z czym odtworzyliśmy wiernie sposób generacji danych w nim opisany. Po tym zabiegu udało nam się odtworzyć dość dobrze oryginalne wyniki. Poniższe wykresy przedstawiają kolejno: ilość stworzonych replik (średnia ilość replik na obiekt), oraz zysk (jaki ułamek kosztu całkowitego pozwoliło zaoszczędzić stworzenie replik) przy stałej ilości obiektów (150), i zmiennej ilości hostów. Są analogiczne do tych umieszczonych w artykule bazowym, brak jednak wykresu obrazującego zysk przy stałej ilości hostów i zmiennej ilości obiektów, ze względu na bardzo duży czas obliczeń (wada języka Python). Algorytm genetyczny skonfigurowany był na stworzenie 10 pokoleń.

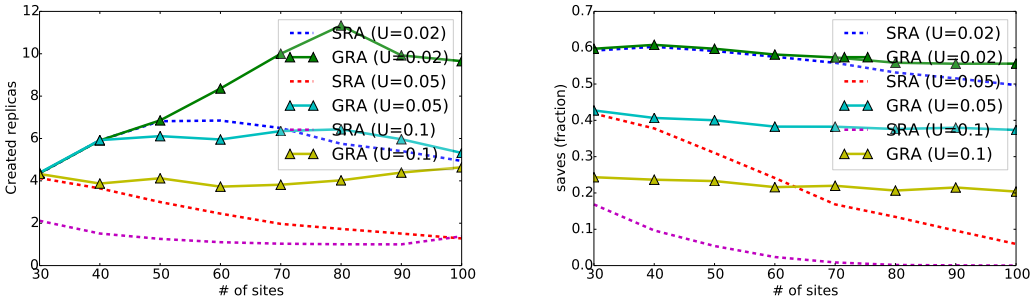


(a) ilość stworzonych replik (b) skalowane

Rysunek 1: Wyniki dla algorytmu z artykułu

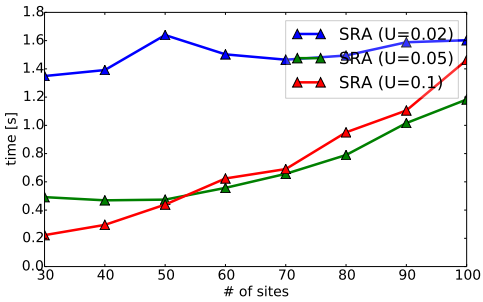
Następnie przystąpiliśmy do stworzenia wielopopulacyjnej wersji algorytmu opartej na modelu wyspowym. Ze względu na stosunkowo duże czasy obliczeń, by w rozsądnym czasie uzyskać wyniki dla odpowiednio dużej liczby wysp, zdecydowaliśmy na zaimplementowanie algorytmu w wersji rozproszonej. Używany przez nas framework PyEvolve posiada wbudowaną obsługę wielopopulacyjności i migracji osobników pomiędzy populacjami, opartą na MPI, jednak na chwilę obecną zdaje się być dość mało dojrzała, i niezbyt dobrze przetestowana – doprowadzenie naszej implementacji do działania wymagało pewnej ingerencji w odpowiadające za to mechanizmy platformy.

Testy wersji wielopopulacyjnej przeprowadzone zostały na komputerze Zeus, z użyciem 12 rdzeni (12 populacji, po jednej na instancję programu), zasymulowane zostało 20 pokoleń.

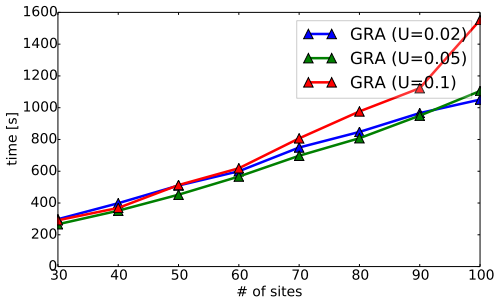


(a) ilość stworzonych replik (b) skalowane

Rysunek 2: Wyniki dla algorytmu wyspowego



(a) ilość stworzonych replik



(b) skalowane

Rysunek 3: Czasy działania algorytmu wyspowego

Jak widać na powyższych wykresach, zastosowanie algorytmu wielopopulacyjnego nie przyniosło zauważalnej poprawy w otrzymanych rozwiązaniach.

[coś o próbach zrobienia

7 Podsumowanie

8 Możliwe kierunki rozwoju

Jakkolwiek algorytm wyspowy nie przyniósł żadnej obserwowalnej poprawy jakości otrzymywanych rozwiązań, w pewnym stopniu może być to spowodowane małym zróżnicowaniem populacji na poszczególnych wyspach. Populacje wybierane są niezależnie przy użyciu tego samego schematu, i są raczej na tyle duże (80 osobników), że trudno się spodziewać, by istotnie się od siebie różniły. Niewykluczone, że stworzenie populacji bardziej zróżnicowanych (poprzez modyfikację rozkładów, z których są losowane –tj. przez użycie różnych sposobów ich inicjalizowania) przyniosłoby lepszy efekt.

Jednym z pomysłów, które ostatecznie nie zostały wprowadzone w życie, była relaksacja operacji genetycznych (mutacji i krzyżowania), polegająca na tym, by wymuszanie poprawności rozwiązań niejako przenieść do etapu ewaluacji – dopuszczać wszystkie stworzone osobniki, jednak stosować kary przy ewaluacji, tak, by osobnikami o największym fitnessie były osobniki z poprawnym genotypem, jednak by zwiększyć „mobilność" populacji – umożliwić zmiany genotypu, które bez relaksacji nie byłyby możliwe, i tym samym być może pozwolić na odkrycie trudnych do znalezienia minimów.

Otwarta pozostaje kwestia znalezienia lepszych operatorów mutacji i krzyżowania, bardziej dostosowanych do rozpatrywanej przestrzeni stanów. Był to jeden z pierwszych pomysłów, jednak nie osiągnęliśmy w tym kierunku żadnych postępów. Znalezienie sensownych operatorów wewnętrznych (nie wychodzących poza – dość nieregularną – przestrzeń stanów) pozwoliłoby uprościć algorytm, a ze względu na pewną „kompatybilność" z przestrzenią stanów być może także polepszyć znajdowane przez ich użycie rozwiązania. Wydaje się to być jednak zadanie stosunkowo trudne, i trudno na chwilę obecną powiedzieć, jakiego rodzaju operacji można by szukać.

Przemyśleć należałoby kwestię generowania danych testowych. Nasze początkowe próby przeprowadzane były na danych generowanych inaczej, niż w artykule, jednak wróciliśmy do niego by odtworzyć wyniki i przy nim pozostaliśmy. Nie jest on jednak w naszym odczuciu idealny. W szczególności obiekcje budzić może topologię połączeń – w artykule bazowym odległości $C(i, j)$ pomiędzy poszczególnymi hostami są losowane z rozkładem jednostajnym ze zbioru $\{1, \ldots, 10\}$. Najbardziej oczywistym problemem zdaje się być pogwałcenie nierówności trójkąta – C nie jest metryką. Trudno powiedzieć, w jaki sposób wpływa to na używany algorytm, jednak wydaje się prawdopodobnym, że mogą istnieć jakieś zmiany/ulepszenia, dla których jest to istotne.

Literatura

[1] T. Loukopoulos I. Ahmad. *Static and Adaptive Data Replication Algorithms for Fast Information Access in Large Distributed Systems*

[2] Biblioteka PyEvolve, http://pyevolve.sourceforge.net/0_6rc1/

[3] Biblioteka mpi4py, http://mpi4py.scipy.org/