**Задоволення розподілених обмежень**

Обговоримо ситуації колаборації, в яких агенти співпрацюють для досягнення спільної мети. Цю мету можна розглядати як спільну між агентами або, альтернативно, як мету центрального дизайнера, який розробляє різних агентів. Звичайно, якщо такий конструктор існує, виникає природне запитання, чому має значення наявність кількох агентів; їх можна розглядати лише як кінцеві датчики і ефектори для виконання плану, розробленого дизайнером. Однак існують ситуації, коли проблему потрібно вирішувати розподіленим способом, або через те, що центральний контролер неможливий, або тому, що хтось хоче раціонально використовувати розподілені ресурси. Хорошим прикладом є сенсорна мережа. Такі мережі складаються з кількох процесорів, кожен з яких має можливості локального датчика, обмежену потужність обробки, обмежене джерело живлення та обмежену пропускну здатність зв’язку. Незважаючи на ці обмеження, такі мережі прагнуть надавати деякі глобальні послуги. Отже, нехай маємо приклад сенсорної мережі, яка використовується для моніторингу навколишніх величин, таких як вологість, температура та тиск в офісному середовищі. Кожен датчик може контролювати лише свою локальну область і, аналогічно, може спілкуватися лише з іншими датчиками поблизу нього. Питання полягає в тому: за яким алгоритмом мають працювати окремі датчики, щоб центр міг скласти надійну глобальну картину?

Зосередимося на алгоритмах розподіленого вирішення проблеми з області штучного інтелекту.

Спершу розглянемо розподілене задоволення обмежень, де агенти намагаються розподіленим способом знайти можливе рішення проблеми з глобальними обмеженнями.

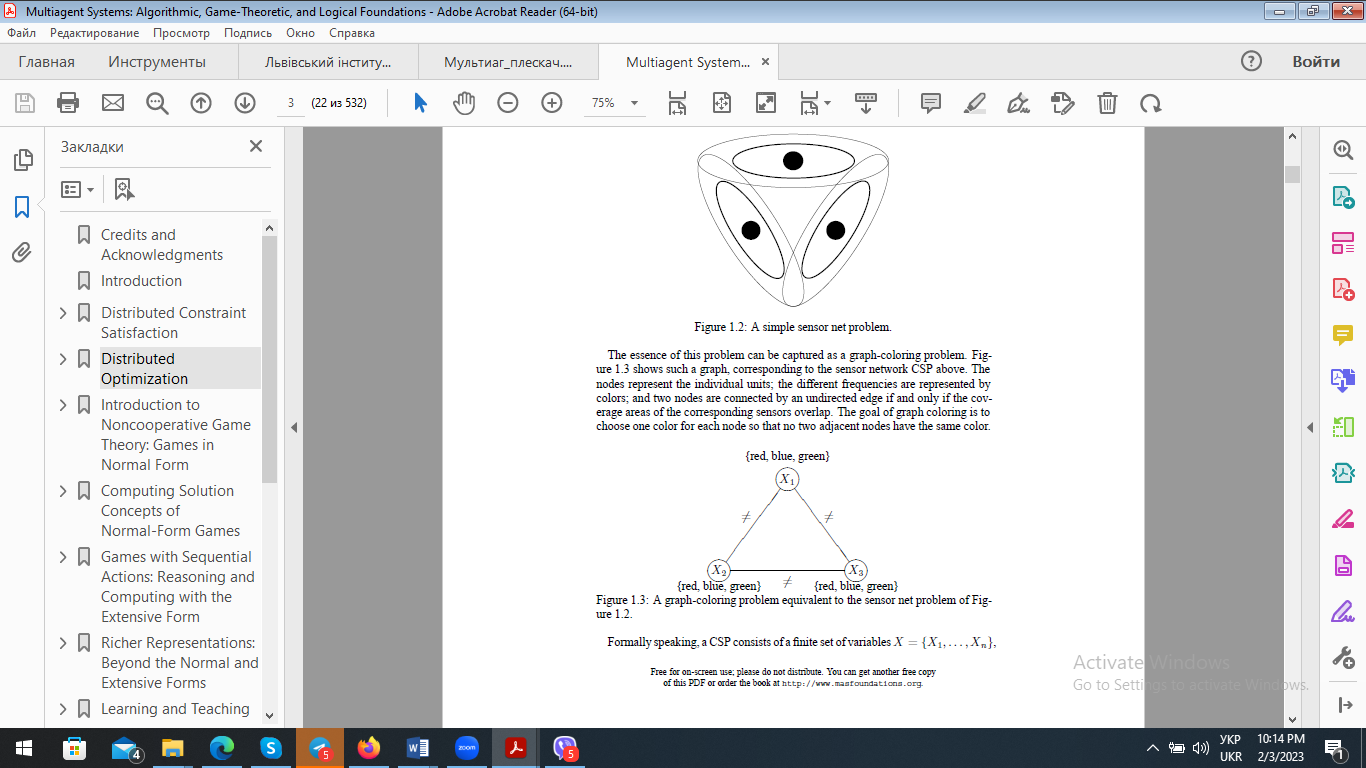
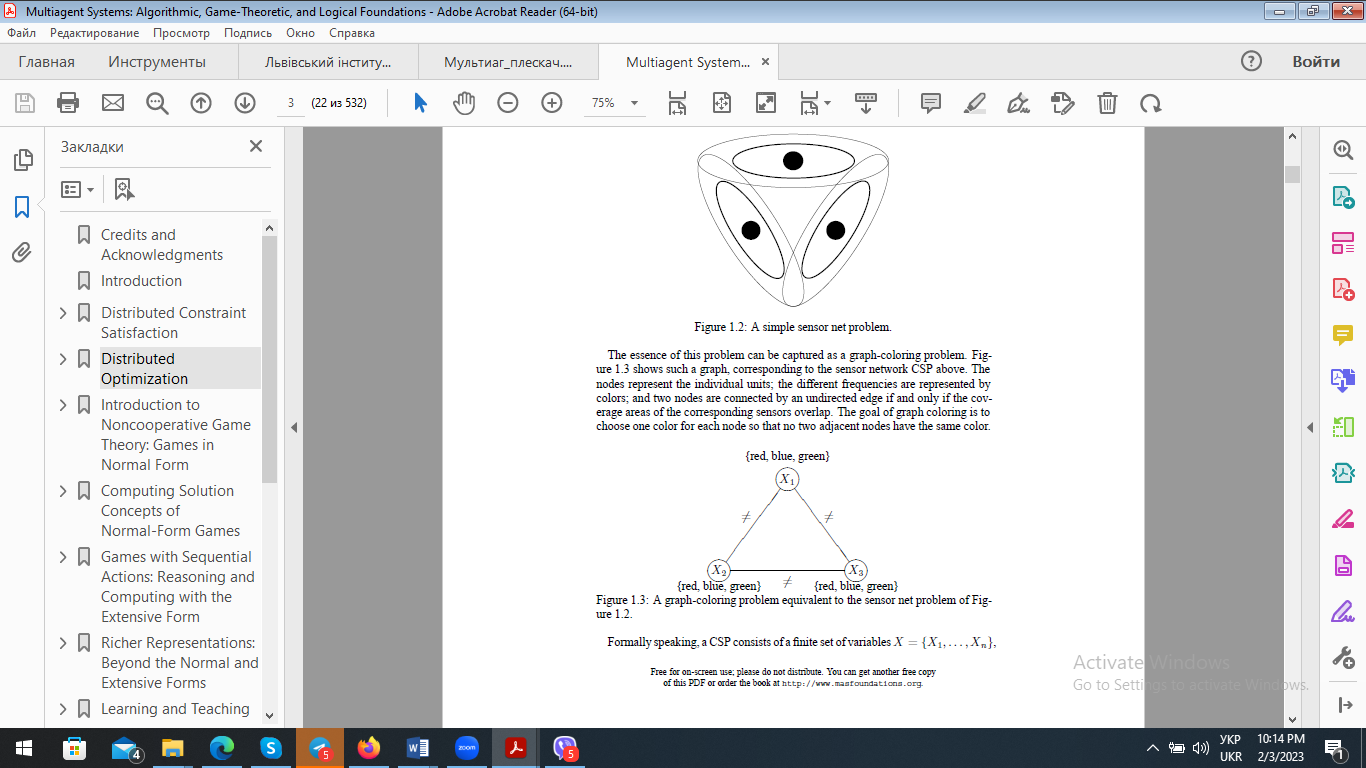
Потім розглянемо агентів, які намагаються не тільки задовольнити обмеження, але й оптимізувати деякі цільові функції, що підлягають цим обмеженням.

*Визначення задач задоволення розподілених обмежень*

Задача задоволення обмежень (CSP, *constraint satisfaction problem*) визначається набором змінних, завданням задоволення доменів для кожної зі змінних і обмеженнями на значення, які змінні можуть приймати одночасно.

Сутність алгоритмів задоволення обмежень полягає в тому, щоб призначити значення змінним у спосіб, який узгоджується з усіма обмеженнями, або визначити, що такого призначення не існує.

Методи задоволення обмежень застосовуються в різних сферах, включаючи машинний зір, обробку природної мови, доведення теорем, ***планування та розклад*** і т.д. Наведемо простий приклад із сенсорних мереж. На рис. 1.2 зображено фрагмент із трьома датчиками сценарію. Кожен із датчиків має певний радіус, який у поєднанні з перешкодами в навколишньому середовищі створює певну зону покриття. Ці зони покриття показані еліпсами на рис. 1.2. Як видно, деякі зони покриття збігаються. Розглянемо конкретну проблему в цій ситуації. Припустимо, що кожен датчик може вибрати одну з трьох можливих радіочастот. Усі частоти працюють однаково добре, якщо два датчики з зонами охоплення, що перекриваються, не використовують однакову частоту. Питання полягає в тому, які алгоритми повинні використовувати датчики для вибору своїх частот, припускаючи, що це рішення не може бути прийняте централізовано.

Суть цієї проблеми можна описати як задачу розфарбування графа. На рис. 1.3 показаний граф, що відповідає нашій сенсорній мережі CSP. Вузли представляють окремі одиниці; різні частоти представлені кольорами; і два вузли з'єднані неорієнтованим ребром тоді і тільки тоді, коли зони покриття відповідних датчиків перекриваються. Мета фарбування графа — вибрати один колір для кожного вузла, щоб жодні сусідні вузли не мали однакового кольору. Формально кажучи, CSP складається з кінцевого набору змінних X = {X1, …, Xn}, області D*i* для кожної змінної X*i* та набору обмежень {C1, . . . ,См}. Хоча загалом CSP допускають нескінченні домени, ми припускаємо, що всі домени обмежені. У наведеному прикладі фарбування графу – 3 змінні, і кожна з них має той самий домен {червоний, зелений, синій}. Кожне обмеження є предикатом деякої підмножини змінних, скажімо, Xi1, . . . ,Xij; предикат визначає відношення, яке є підмножиною декартового добутку Di1 × · · · × Dij . Кожне таке обмеження обмежує значення, які можуть бути одночасно призначені змінним, що беруть участь у обмеженні. Обмежимо зараз обговорення бінарними обмеженнями, кожне з яких обмежує рівно дві змінні. Наприклад, у випадку розфарбовування карти, кожне обмеження «не рівно» застосовується до двох вузлів.

Враховуючи підмножину S змінних, екземпляр S є призначенням унікального значення домену для кожної змінної в S; воно є законним, якщо воно не порушує жодних обмежень, які згадують лише змінні в S. Рішення для мережі є законним екземпляром усіх змінних. Типові завдання, пов’язані з мережами обмежень, полягають у тому, щоб визначити, чи існує рішення, знайти одне або всі рішення, визначити, чи можна поширити юридичну реалізацію деяких змінних на рішення тощо.

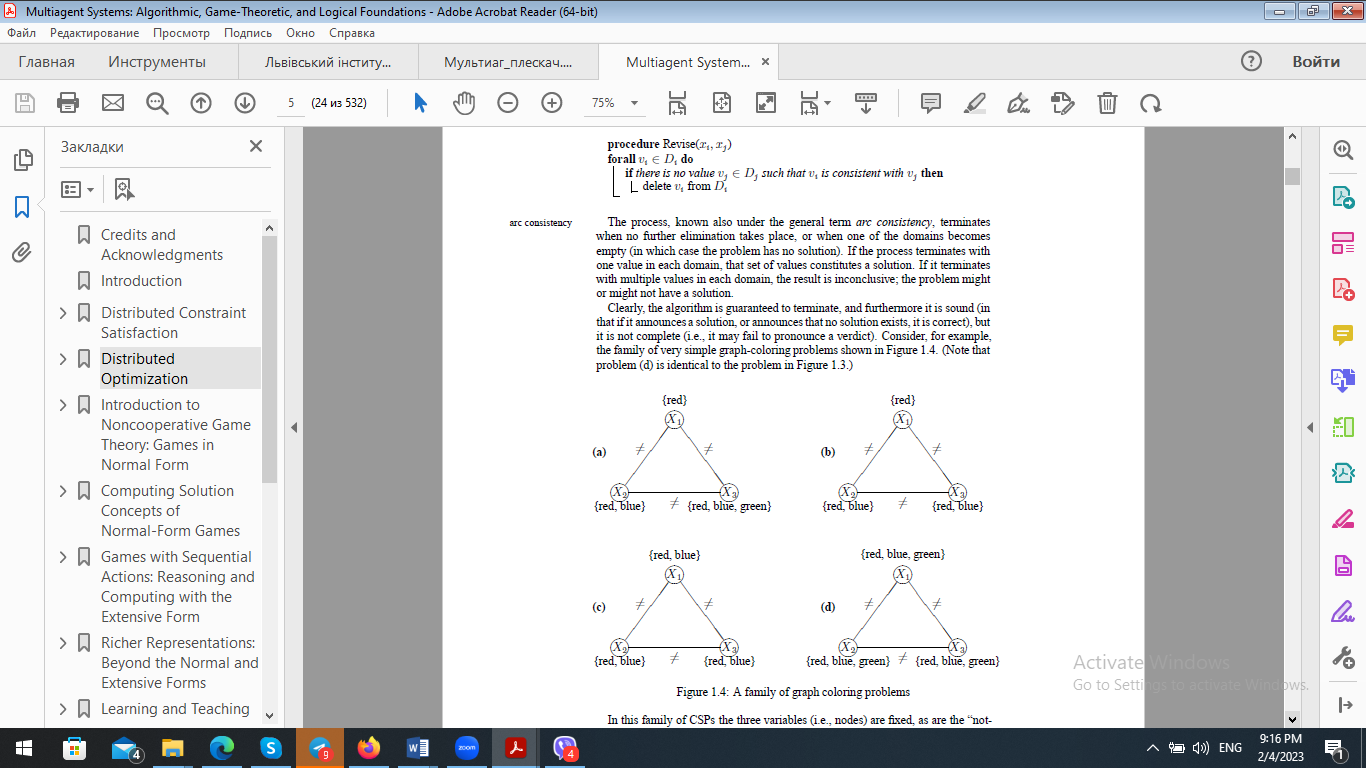
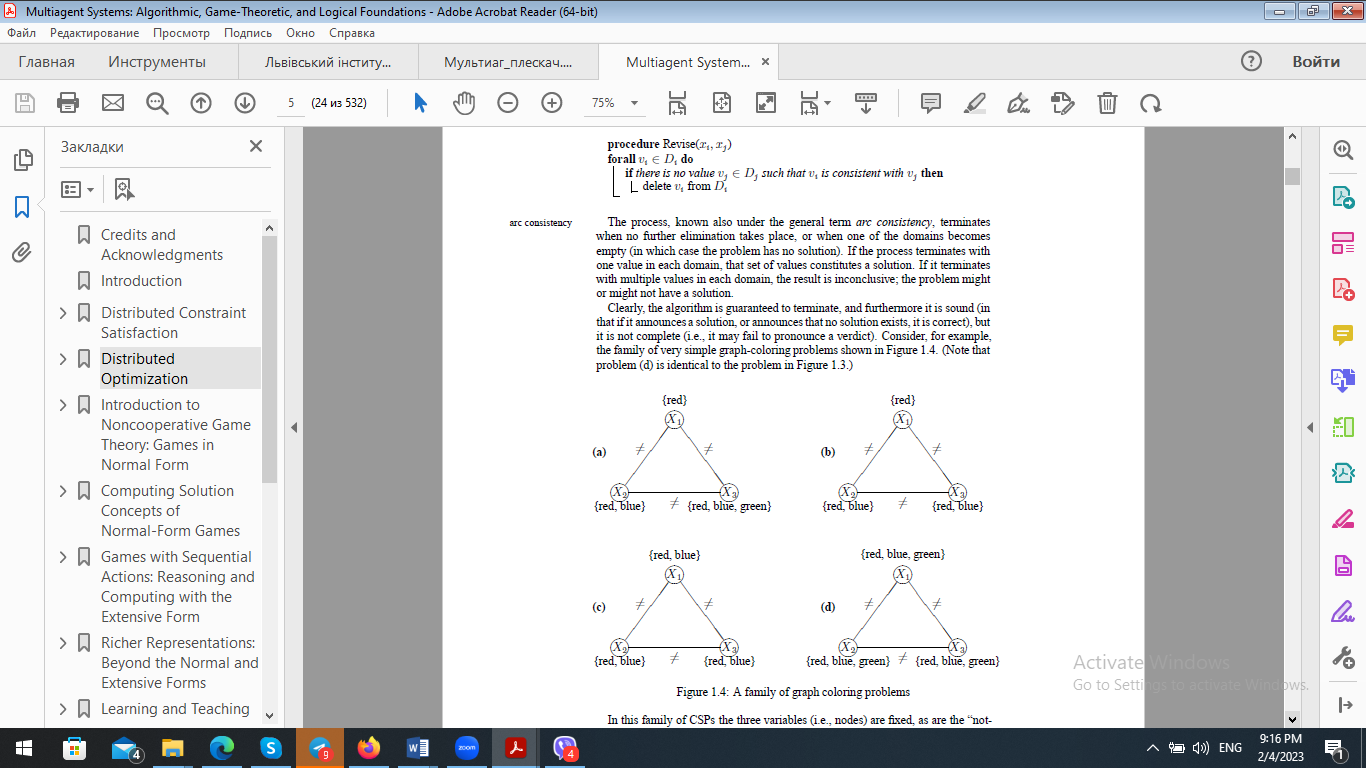
Зосередимося на найпоширенішому завданні, яке полягає в тому, щоб знайти одне рішення CSP або довести, що такого не існує. У розподіленому CSP кожна змінна належить іншому агенту. Мета все ще полягає в тому, щоб знайти глобальне призначення змінної, яке відповідає обмеженням, але кожен агент вирішує значення своєї власної змінної з відносною автономією. Хоча кожен агент не має можливості глобального перегляду, кожен агент може спілкуватися зі своїми сусідами в графі обмежень. Розподілений алгоритм для вирішення CSP включає кожного агента в якомусь протоколі, який поєднує локальні обчислення зі зв'язком з його сусідами. Хороший алгоритм гарантує, що такий процес завершиться юридичним рішенням (або усвідомленням того, що юридичного рішення не існує) і робить це швидко. Розглянемо 2 типи алгоритмів. Алгоритми 1-ого типу втілюють підхід найменшого зобов’язання та намагаються виключити неможливі значення змінних без втрати будь-яких можливих рішень. Алгоритми 2-ого типу більш авантюрні і вибирають орієнтовні значення змінних, повертаючись назад, коли ці вибори виявляються невдалими. В обох випадках ми припускаємо, що зв’язок між сусідніми вузлами ідеальний, але це не стосується його часу; повідомлення можуть займати більше або менше часу без причини. Однак ми припускаємо, що якщо вузол *i* надсилає кілька повідомлень до вузла *j*, ці повідомлення надходять у тому порядку, в якому вони були надіслані.

**1. Алгоритми скорочення домену**

Згідно з алгоритмами скорочення домену, вузли спілкуються зі своїми сусідами, щоб видалити значення зі своїх доменів.

Розглянемо два таких алгоритми.

1. **Алгоритм фільтрації.** Кожен вузол передає інформацію про фільтрацію свого домену своїм сусіднім, усуває зі свого домену значення, які не відповідають значенням, отриманим від сусідів, і процес повторюється. Зокрема, кожен вузол x*i* з доменом D*i* неодноразово виконує процедуру перегляду Revise(x*i*, x*j*) для кожного сусіда x*j*.

  Процес, відомий під загальним терміном *узгодженість дуги*, припиняється, коли не відбувається подальшого усунення, або коли одна з областей стає порожньою (тоді проблема не має розв’язку). Якщо процес завершується одним значенням у кожній області, цей набір значень є рішенням. Якщо він завершується декількома значеннями в кожній області, результат непереконливий; проблема може мати або не мати рішення. Зрозуміло, що алгоритм гарантовано припинить роботу, і, крім того, він правильний (якщо він оголошує рішення або оголошує, що рішення не існує, він правильний), але він не є повним (тобто він може не винести вердикт). Розглянемо, наприклад, сімейство дуже простих задач розфарбовування графів на рис. 1.4. (задача (d) ідентична задачі на рис. 1.3.).

У цьому сімействі CSP три змінні (тобто вузли) є фіксованими, як і обмеження «нерівності» між ними. Що не є фіксованим, так це домени змінних. Розглянемо 4 приклади на рис. 1.4. (a) Спочатку, коли вузли спілкуються один з одним, лише повідомлення x1 призводять до будь-яких змін. Зокрема, коли x2 або x3 отримують повідомлення x1, вони видаляють червоний колір зі своїх доменів, в результаті чого D2 = {blue} і D3 = {blue, зелений}. Потім, коли x2 передає свій новий домен x3, x3 ще більше зменшує свій домен до {зеленого}. На цьому етапі подальші зміни не відбуваються, і алгоритм завершується правильним рішенням. (b) Алгоритм починається, як і раніше, але як тільки x2 і x3 отримують повідомлення від x1, кожен з них зменшує свої домени до {blue}. Тепер, коли вони оновлюють один одного щодо своїх нових доменів, кожен з них скорочує свої домени до {}, порожнього набору. На цьому етапі алгоритм завершує роботу та правильно повідомляє, що рішення не існує. (c) У цьому випадку початковий набір повідомлень не дає скорочення в жодному домені. Алгоритм припиняє роботу, але всі вузли мають кілька значень. І тому алгоритм не може показати, що проблема надмірно обмежена і не має розв’язку. (d) Фільтрування також може бути невдалим, якщо існує рішення. З тих самих причин, що й у прикладі (c), алгоритм не може показати, що в цьому випадку проблема дійсно має рішення. Загалом, фільтрування є дуже слабким методом і, у кращому випадку, використовується як етап попередньої обробки для більш складних методів. Алгоритм безпосередньо базується на понятті *розділення одиниць* із пропозиційної логіки. Роздільна здатність одиниці є наступним правилом висновку:

A1

￢(A1 ∧ A2 ∧ · · · ∧ An)

￢(A2 ∧ · · · ∧ An)

Щоб побачити, як алгоритм фільтрації відповідає роздільній здатності одиниці, ми повинні спочатку записати обмеження у вигляді заборонених комбінацій значень, які називаються Nogoods. Наприклад, обмеження, що x1 і x2 не можуть мати значення «червоний», призведе до пропозиційного речення ¬(x1 = red∧ x2 = red), яке ми пишемо як Nogood {x1, x2}. У прикладі (b) на рис. 1.4 агент X2 оновив свій домен на основі оголошення агента X1 про те, що x1 = red і Nogood {x1 = red, x2 = red}

x1 = red

￢(x1 = red ∧ x2 = red)

￢(x2 = red)

Роздільна здатність одиниць є слабким правилом висновку, тому не дивно, що алгоритм фільтрації також слабкий. *Гіперроздільна здатність* є узагальненням одиничної роздільної здатності та має такий вигляд:

A1 ∨ A2 ∨ · · · ∨ Am

￢(A1 ∧ A1,1 ∧ A1,2 ∧ · · · )

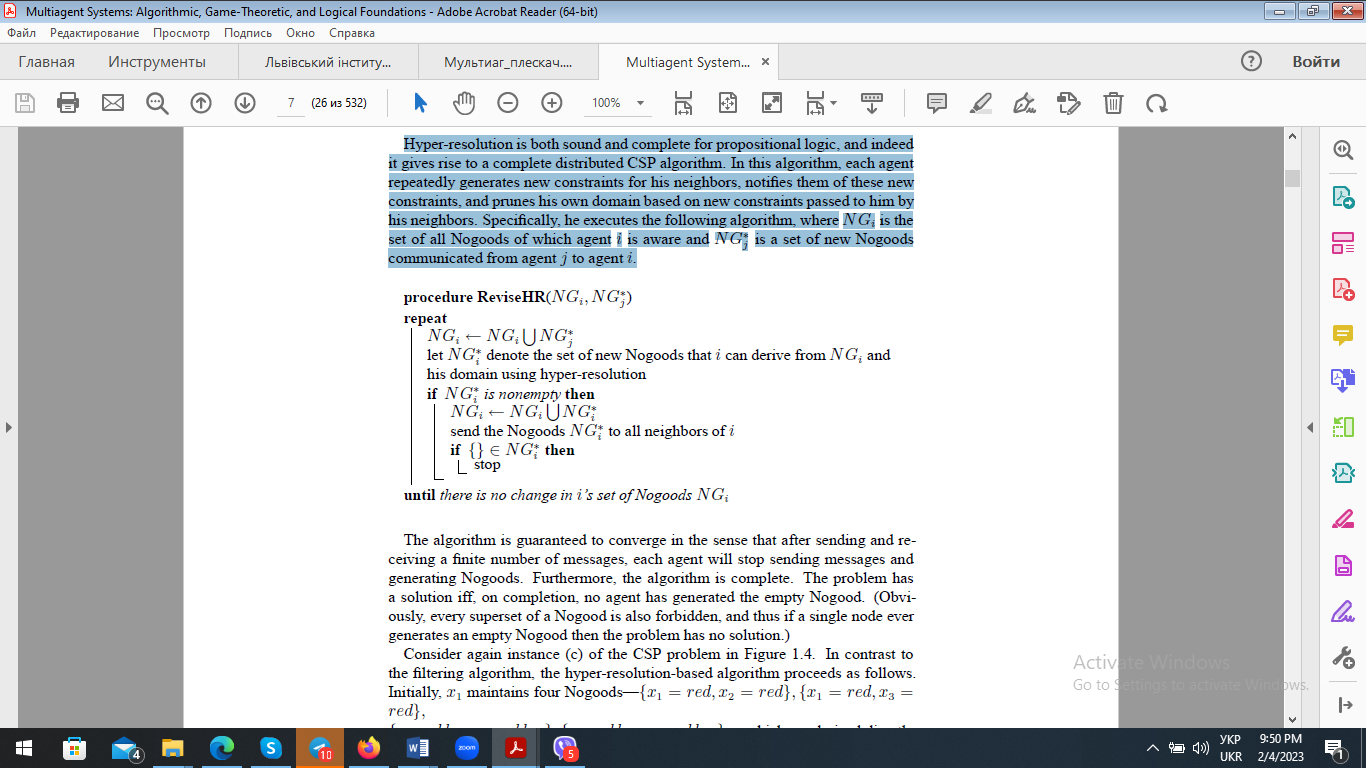
￢(A2 ∧ A2,1 ∧ A2,2 ∧ · · · )

...

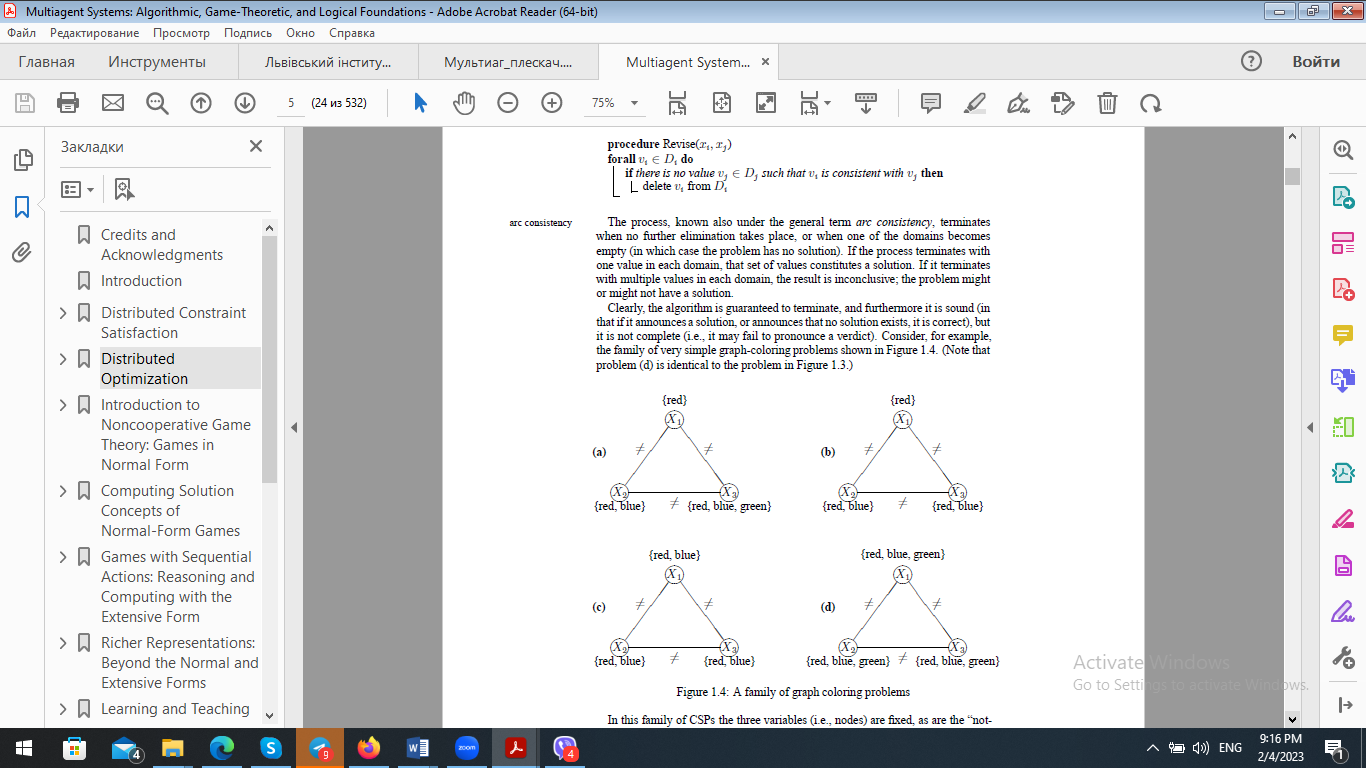
￢(Am ∧ Am,1 ∧ Am,2 ∧ · · · )

￢(A1,1 ∧ · · · ∧ A2,1 ∧ · · · ∧ Am,1 ∧ · · · )

2. Гіперроздільна здатність є надійною та повною для пропозиційної логіки, і справді вона дає початок повному розподіленому алгоритму CSP. У цьому алгоритмі кожен агент неодноразово створює нові обмеження для своїх сусідів, повідомляє їх про ці нові обмеження та відрізає свій власний домен на основі нових обмежень, переданих йому його сусідами. Зокрема, він виконує наступний алгоритм, де NG*i* — це набір усіх Nogoods, про які агент *i* знає, а NG∗*j* — це набір нових Nogoods, переданих від агента *j* до агента *i*.



Алгоритм гарантовано сходиться в тому сенсі, що після надсилання та отримання кінцевої кількості повідомлень кожен агент припинить надсилати повідомлення та генерувати Nogoods. Крім того, алгоритм завершений. Проблема має рішення, якщо після завершення жоден агент не згенерував порожній Nogood. (Очевидно, що кожна надмножина Nogood також заборонена, і тому, якщо один вузол коли-небудь генерує порожній Nogood, проблема не має вирішення.) Знову розглянемо приклад (c) проблеми CSP на рис. 1.4. На відміну від алгоритму фільтрації, алгоритм на основі **гіперроздільності** працює наступним чином.



Спочатку x1 підтримує чотири Nogoods — {x1 = red, x2 = red}, {x1 = red, x3 = red}, {x1 = blue, x2 = blue}, {x1 = blue, x3 = blue} — які є похідними безпосередньо з обмежень, що включають x1. Крім того, x1 має прийняти одне зі значень у своїй області, тому x1 = red ∨ x1 = blue. Використовуючи гіперроздільну здатність, x1 може вважати:

x1 = red ∨ x1 = blue

￢(x1 = red ∧ x2 = red)

￢(x1 = blue ∧ x3 = blue)

￢(x2 = red ∧ x3 = blue)

Таким чином, x1 створює новий Nogood {x2 = red, x3 = blue}; подібним чином він також може побудувати Nogood {x2 = blue, x3 = red}. Потім x1 надсилає обидва Nogoods своїм сусідам x2 і x3. Використовуючи свій домен, існуючий Nogood і один із цих нових Nogoods, x2 може міркувати:

x2 = red ∨ x2 = blue

￢(x2 = red ∧ x3 = blue)

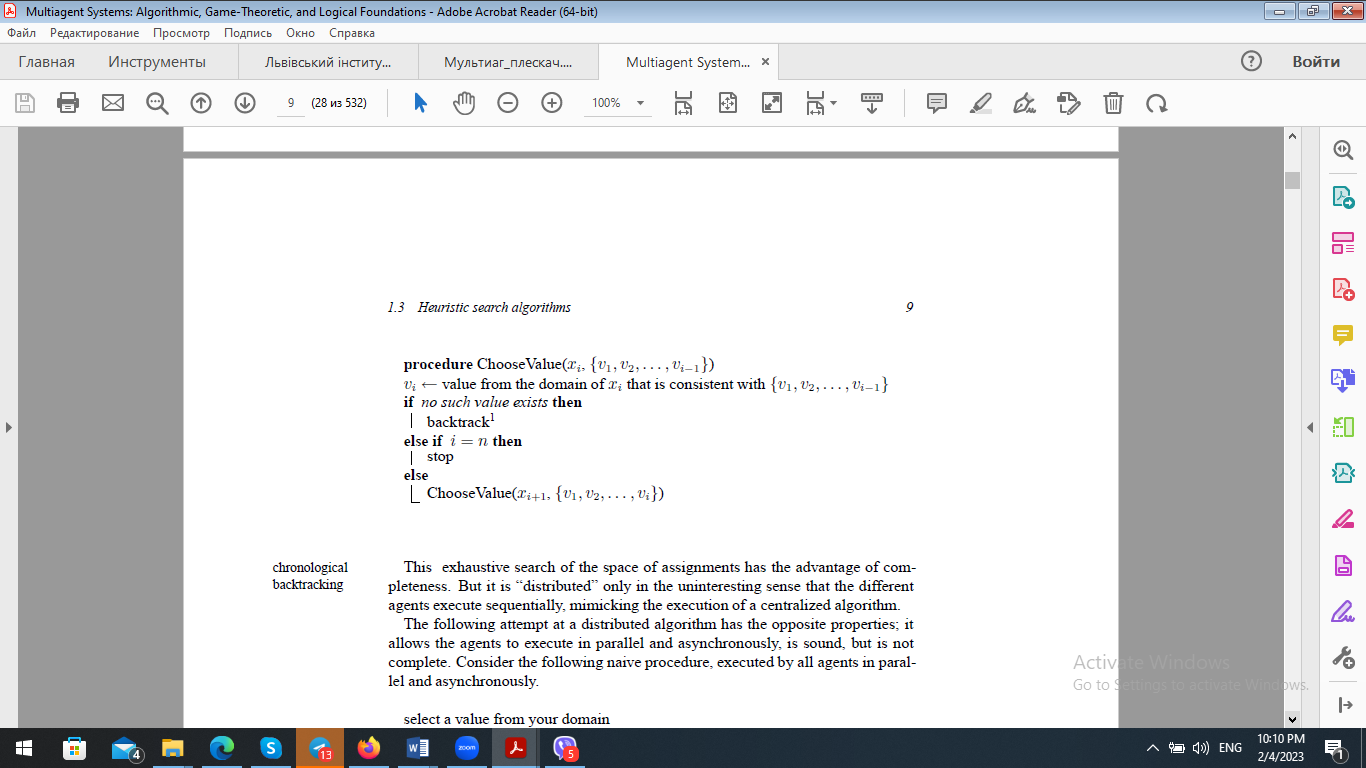
￢(x2 = blue ∧ x3 = blue)

￢(x3 = blue)

Використовуючи інший новий Nogood з x1, x2 також може побудувати Nogood {x3 = red}. Ці два одиночних Nogood передаються x3 і дозволяють йому згенерувати порожній Nogood. Це доводить, що проблема не має рішення. Цей приклад, демонструючи більшу потужність алгоритму на основі гіперроздільності порівняно з алгоритмом фільтрації, також виявляє його непрактичність; кількість створених Nogoods може стати некеровано великою (справді, описали лише мінімальну кількість Nogoods, необхідну для отримання порожнього Nogood; багато ще буде створені, коли всі агенти оброблятимуть повідомлення один одного паралельно). Таким чином, ситуація, в якій ми опинилися полягає в тому, що у нас є один алгоритм, який є надто слабким, а інший – непрактичним. Проблема полягає в природі цих алгоритмів з найменшим зобов'язанням; вони обмежені видаленням лише доведено неможливих комбінацій цінностей. Альтернативою таким «безпечним» процедурам є дослідження підмножини простору, вибір попередніх значень для змінних і повернення назад, коли необхідно. Проте описані алгоритми не є недоречними: алгоритм фільтрації є ефективним етапом попередньої обробки, а алгоритм, який обговорюємо далі, базується на алгоритмі на основі гіперроздільності.

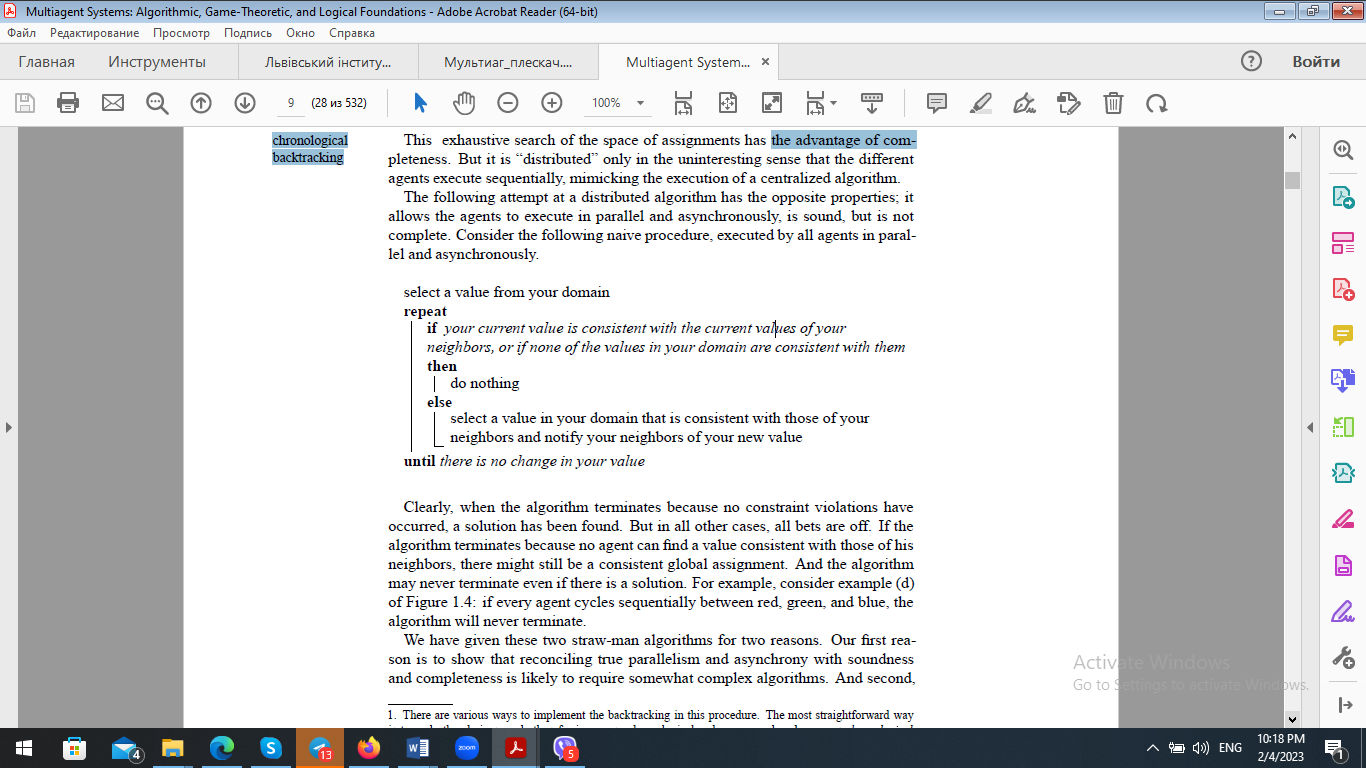
**Евристичні пошукові алгоритми**

Просте централізоване рішення задачі задоволення обмежень (CSP) методом проб і помилок полягає в тому, щоб спочатку впорядкувати змінні (наприклад, за алфавітом). Тоді, враховуючи порядок x1, x2, . . . , xn, викликати процедуру ChooseValue(x1, {}). Процедура ChooseValue визначається рекурсивно таким чином (тут {v1, v2, . . . , vi−1} — набір значень, присвоєних змінним x1, . . . , xi−1):



Цей вичерпний пошук простору призначень має перевагу повноти. Але він «розподілений» лише в тому нецікавому сенсі, що різні агенти виконуються послідовно, імітуючи виконання централізованого алгоритму. Наступна спроба розподіленого алгоритму має протилежні властивості; він дозволяє агентам виконувати паралельно та асинхронно, є надійним, але не повним. Розглянемо наступну наївну процедуру, яка виконується всіма агентами паралельно і асинхронно.

*Хронологічний відкат* (iснують різні способи реалізації зворотного відстеження в цій процедурі. Найпростіший спосіб — скасувати вибір, зроблений до цього моменту, у зворотному хронологічному порядку)

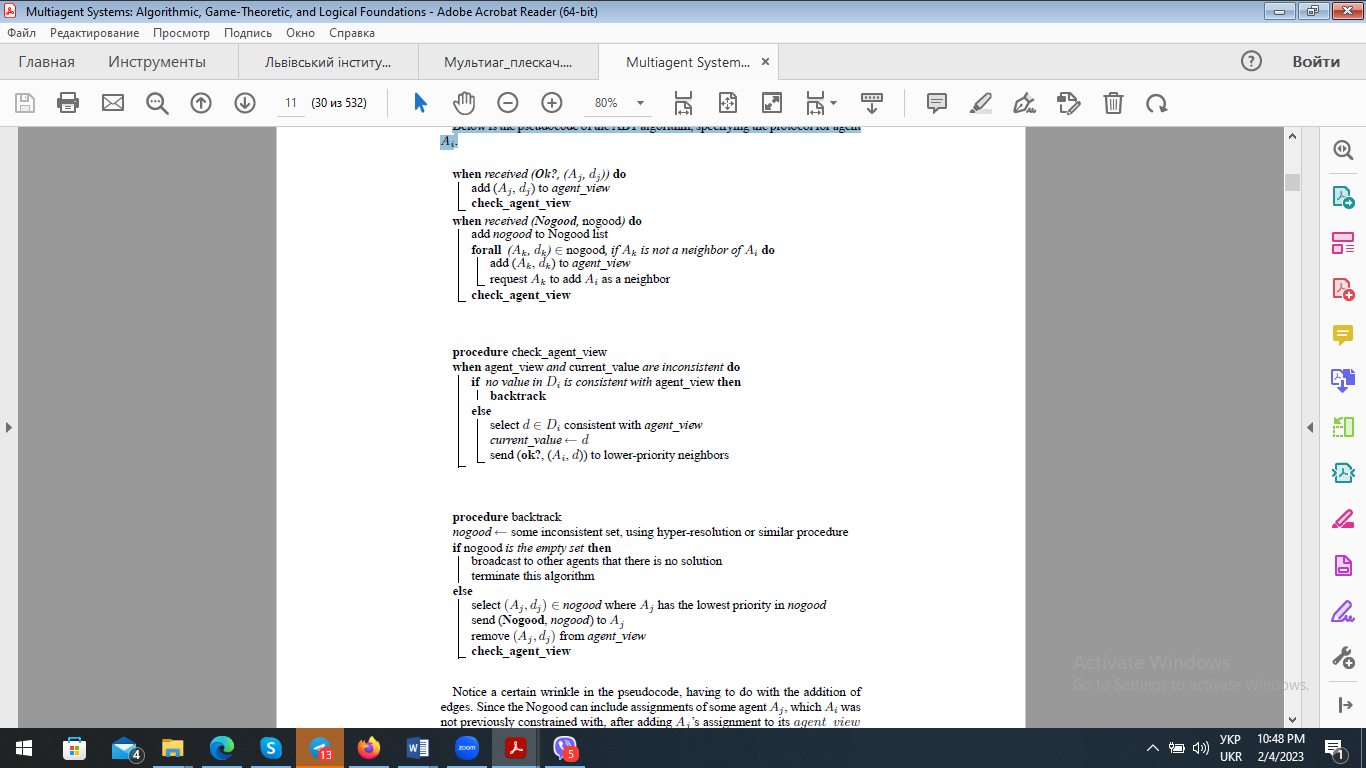


Зрозуміло, що коли алгоритм припиняє роботу через відсутність порушень обмежень, рішення знайдено. Але в усіх інших випадках усі ставки скасовуються. Якщо алгоритм припиняє роботу через те, що жоден агент не може знайти значення, яке відповідає значенням його сусідів, все ще може існувати узгоджене глобальне призначення. І алгоритм може ніколи не припинити роботу, навіть якщо є рішення. Наприклад, у прикладі (d) на рис. 1.4: якщо кожен агент послідовно перемикається між червоним, зеленим і синім, алгоритм ніколи не завершиться.

Надані ці два алгоритми «соломинної людини» з двох причин. Перша причина полягає в тому, щоб показати, що узгодження справжнього паралелізму й асинхронності з надійністю й повнотою, імовірно, потребує складніших алгоритмів. І по-друге, фундаментальний евристичний алгоритм для розподілених CSP — **алгоритм асинхронного відстеження (або ABT, asynchronous backtracking**) — дещо схожий з цими двома алгоритмами. З першого він запозичив поняття глобального загального порядку на агентах. У другого запозичив протокол передачі повідомлень, хоча і більш складний, який спирається на глобальне впорядкування. Опишемо ABT у найпростішому вигляді.

**Алгоритм асинхронного зворотного відстеження** (ABT)

ABT передбачає загальне впорядкування («порядок пріоритетів») для агентів. Кожне бінарне обмеження відоме обом обмеженим агентам і перевіряється в алгоритмі агентом із нижчим з них пріоритетом. Посилання в мережі обмежень завжди спрямовується від агента з вищим пріоритетом до агента з нижчим пріоритетом. Агенти всі одночасно створюють екземпляри своїх змінних і надсилають призначені їм значення агентам, підключеним до них вихідними посиланнями. Усі агенти чекають і відповідають на повідомлення. Після кожного оновлення свого призначення агент надсилає своє нове призначення по всіх вихідних посиланнях. Агент, який отримує призначення (від агента зв'язку з вищим пріоритетом), намагається знайти призначення для своєї змінної, яке не порушує обмеження з призначенням, яке він отримав. ОК? повідомлення — це повідомлення, що містять призначення змінної агента. Коли агент A*i* отримує OK? повідомлення від агента A*j*, A*i* розміщує отримане призначення в структурі даних під назвою agent\_view, яка містить останнє призначення A*i*, отримані від сусідів з вищим пріоритетом, таких як A*j* . Далі А*і* перевіряє, чи його поточне призначення все ще відповідає його agent\_view. Якщо воно узгоджене, A*i* нічого не робить. Якщо ні, тоді А*і* шукає у своєму домені нове узгоджене значення. Якщо він знайде таке, він присвоїть своїй змінній це значення та надішле OK? повідомлення всім нижчим пріоритетним агентам, пов’язаним з ним, інформуючи їх про це значення. Інакше A*i* повертається назад. Операція повернення виконується шляхом надсилання повідомлення Nogood. Nogood — це просто неузгоджене часткове призначення, тобто призначення певних значень деяким змінним, які разом порушують обмеження на ці змінні. У цьому випадку Nogood складається з A*i*-го agent\_view. Nogood надсилається агенту з найнижчим пріоритетом серед агентів, призначення яких включено до неузгодженого кортежу в Nogood. Агент A*i*, який надсилає Nogood-повідомлення агенту A*j*, припускає, що A*j* змінить своє призначення. Тому A*i* видаляє зі свого agent\_view призначення A*j* і робить спробу знайти призначення для змінної A*j*, яке відповідає оновленому agent\_view. Через його залежність від створення набору Nogoods, алгоритм ABT можна розглядати як жадібну версію алгоритму гіперроздільності з попереднього розділу. Там всі можливі Nogoods генеруються кожним агентом і передаються всім сусідам, навіть якщо переважна більшість цих повідомлень не є корисними. В цьому алгоритмі агенти роблять попередній вибір значення для своїх змінних, лише генерують Nogoods, які включають значення, уже згенеровані агентами, розташованими вище за них у порядку, і, що важливо, повідомляють нові значення лише деяким агентам, а нові Nogoods лише одному агенту. Нижче наведено псевдокод алгоритму ABT, що визначає протокол для агента A*i*.



Зверніть увагу на момент в псевдокоді, пов’язаний з додаванням ребер. Оскільки Nogood може включати призначення деякого агенту Aj, які Ai раніше не були обмежені, після додавання призначення Aj до свого agent\_view Ai надсилає повідомлення Aj з проханням додати Ai до свого списку вихідних посилань. Крім того, після додавання посилання Aj надсилає ok? повідомлення Ai кожного разу, коли він перепризначає свою змінну. Після збереження Nogood Ai перевіряє, чи його призначення все ще є відповідним, узгодженим (consistent, не суперечливим). Якщо це так, повідомлення надсилається агенту, від якого отримано Nogood. Це повторне надсилання призначення має вирішальне значення, оскільки, як згадувалося раніше, агент, який надсилає Nogood, припускає, що одержувач Nogood замінює його призначення. Тому він повинен знати, що переуступка все ще дійсна. Якщо старе призначення, яке було заборонено Nogood, є суперечливим, Ai намагається знайти нове призначення подібно до випадку, коли OK? повідомлення отримано.

**Простий приклад.**

Ілюстрація роботи алгоритму ABT над однією з проблем, які зустрічалися вище. Знову розглянемо екземпляр (c) CSP на рис. і припустимо, що агенти розташовані в алфавітному порядку: x1, x2, x3. Спочатку вони вибирають значення навмання; припустимо, що всі вони вибирають синій колір. x1 повідомляє x2 і x3 про свій вибір, і x2 повідомляє x3. Таким чином, локальний вигляд x2 має {x1 = blue}, а локальний вигляд x3 — {x1 = blue, x2 = blue}. x2 і x3 повинні перевіряти узгодженість своїх локальних представлень із власними значеннями. x2 виявляє конфлікт, змінює власне значення на червоний і повідомляє x3. Тим часом x3 також перевіряє узгодженість і так само змінює своє значення на червоне; він, однак, нікого не повідомляє. Потім x3 отримує друге повідомлення від x2 і оновлює свій локальний вигляд до {x1 = blue, x2 = red}. В цьому він не може знайти значення зі свого домену, яке відповідає його локальному погляду, і, використовуючи гіперроздільність, генерує Nogood {x1 = blue, x2 = red}. Він передає цей Nogood x2, агенту з найнижчим рангом, який бере участь у Nogood. Тепер x2 не може знайти значення, яке відповідає його локальному погляду, створює Nogood {x1 = blue} і передає його x1. x1 виявляє невідповідність його поточному значенню, змінює своє значення на червоний і повідомляє нове значення x2 і x3. Тепер процес продовжується, як і раніше; x2 змінює своє значення назад на синє, x3 не знаходить узгодженого значення та генерує Nogood {x1 = red, x2 = blue}, а потім x2 генерує Nogood {x1 = red}. У цей момент x1 має Nogood {x1 = blue}, а також Nogood {x1 = red}, і використовуючи гіперроздільну здатність, він генерує Nogood {}, і алгоритм завершує роботу, визначивши, що проблема не має розв’язку.

Необхідність додавання нових ребер видно на дещо зміненому прикладі, показаному на рис. нижче Як і в попередньому прикладі, тут також x3 генерує Nogood {x1 = blue, x2 = red} і повідомляє x2. x2 не може відновити послідовність, змінивши власну цінність. Однак x1 не є сусідом x2, тому x2 не має значення x1 = blue у своєму локальному поданні та не може надіслати Nogood {x1 = blue} до x1. Отже, x2 надсилає запит x1 додати x2 до його списку сусідів і надіслати x2 його поточне значення. Далі алгоритм продовжується, як і раніше.

